Visualització de models 3D amb correlació d'imatges de la web

Arnau González Massó

Gener de 2008

Índex

1 Introducció

Una de les branques de la informàtica que està evolucionant més ràpidament i que s'està prenent cada vegada més en consideració per la indústria i el mercat són els gràfics per ordinador. Degut al gran relleu que estan prenent els videojocs actualment, la gran utilització d'efectes digitals en les pel·lícules i els avantatges que proporcionen les aplicacions gràfiques a la medicina, arquitectura i altres, aquesta branca ha pres un posició més que destacada.

Concretament la ràpida evolució del maquinari per potenciar el processat gràfic i la millora de les tècniques i eines de desenvolupament fan que els entorns tridimensionals i plens d'imatges es facin cada cop més habituals en tot tipus de dispositiu. Així no és estrany trobar aplicacions 3D amb finalitats ben diverses i innovadores.

En aquest projecte s'han pres el gràfics per ordinador com a element principal i s'ha treballat en alguns dels seus aspectes més destacats.

1.1 Motivació

Degut a la presència cada cop més gran dels gràfics per ordinador en la nostra societat i a l'atractiu que aporten, aquest projecte s'ha enfocat a descobrir-ne alguns dels seus secrets, aprofundir en el seu desenvolupament i mirar de donar-los-hi una aplicació innovadora.

1.2 Objectius

L'objectiu principal en aquest projecte era partir dels coneixements de gràfics adquirits a través de les diverses assignatures de la carrera i seguir aprofundint en el seu estudi, desenvolupament i aplicabilitat.

Així doncs es va decidir crear un visualitzador de models 3D com ja s'-havia treballat anteriorment però aportant-li conceptes més avançats com

Figura 1: Els gràfics 3D avui en dia

el de la optimització del pintat de la geometria. Això serviria com a base per a construir una aplicació que ens permetés fer comparacions entre diferents vistes dels models 3D i imatges obtingudes de la Web, introduint així també el concepte de gràfics 2D.

L'aplicació final doncs hauria de permetre visualitzar models 3D i interactuar amb ells per obtenir diverses perspectives. A més hauria de ser capaç de comparar les perspectives del model amb imatges mitjançant diversos algorismes de comparació.

Alguns d'aquests algorismes formen part de tècniques bastant noves, així que més que orientar l'aplicació a una utilitat concreta, l'interès estarà en avaluar els resultats aportats per aquests algorismes.

1.3 Gràfics per ordinador

Molt lluny queden ja els temps en que la informàtica eren un munt de línies de text sobre un fons negre. Aviat es va veure que si es volia que fos accessible per la gent no entesa se li havia de donar un aspecte més atractiu i fàcil d'utilitzar. D'aquesta idea apareix la informàtica gràfica i amb ella les interfícies de botons, les imatges de tot tipus i posteriorment els gràfics en tres dimensions.

Figura 2: La informàtica abans dels gràfics

Figura 3: Imatges planes, interfícies gràfiques i escenes 3D

Amb el temps aquests aspectes han anat evolucionant i les eines per treballar amb ells s'han anat multiplicant i perfeccionant. Avui en dia disposem d'una gran diversitat de programari per crear interfícies gràfiques en diferents llenguatges, visualitzar i treballar amb imatges planes, i construir i interactuar amb entorns 3D.

1.3.1 Entorn 3D amb *OpenGL*

Per crear entorns 3D necessitem donar-li instruccions al maquinari gràfic perquè faci càlculs geomètrics, processi colors i sigui capaç de mostrar-nos els seus resultats a través d'un dispositiu com ara un monitor.

Actualment disposem de llibreries gràfiques que ens proporcionen els recursos per demanar-li a un ordinador que ens faci aquesta feina. Les llibreries s'encarreguen de treballar a baix nivell, en col·laboració amb la màquina, i permeten als desenvolupadors centrar-se en la part conceptual i creativa.

Les llibreries gràfiques amb més presència actualment són *DirectX*, creada per *Microsoft*; i *OpenGL*, guiada pel consorci independent *OpenGL Architecture Review Board* i de la qual existeixen diverses implementacions lliures i propietàries. Degut bàsicament al fet de ser un estàndard obert i a la seva potència i facilitat d'ús, s'ha escollit *OpenGL* per al nostre desenvolupament. *OpenGL*, a més, té implementacions per gairebé totes les plataformes (Linux, Windows, MacOS, etc.) cosa que el fa l'estàndard més utilitzat en la programació d'aplicacions 3D.

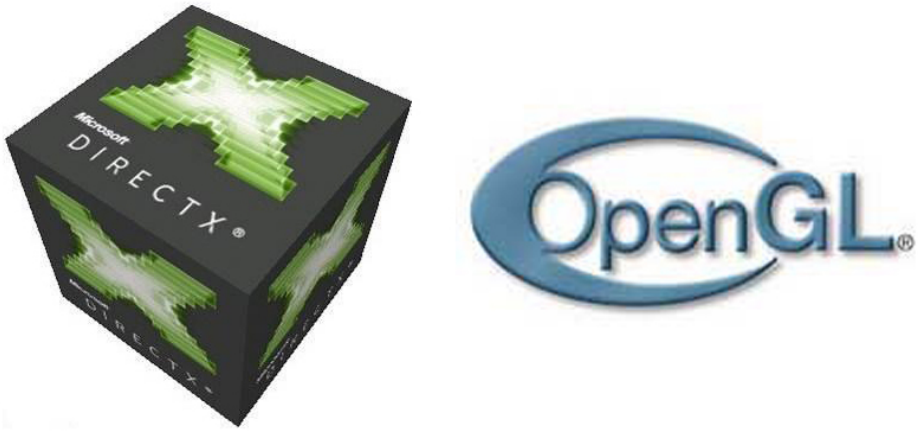


Figura 4: Logos de DirectX i OpenGL

OpenGL és una llibreria gràfica definida per l'especificació *The OpenGL Graphics System: A Specification* [2], actualment en la versió 2.1, i els programes *OpenGL* utilitzen implementacions d'aquesta especificació per representar geometria 2D i 3D i imatges.

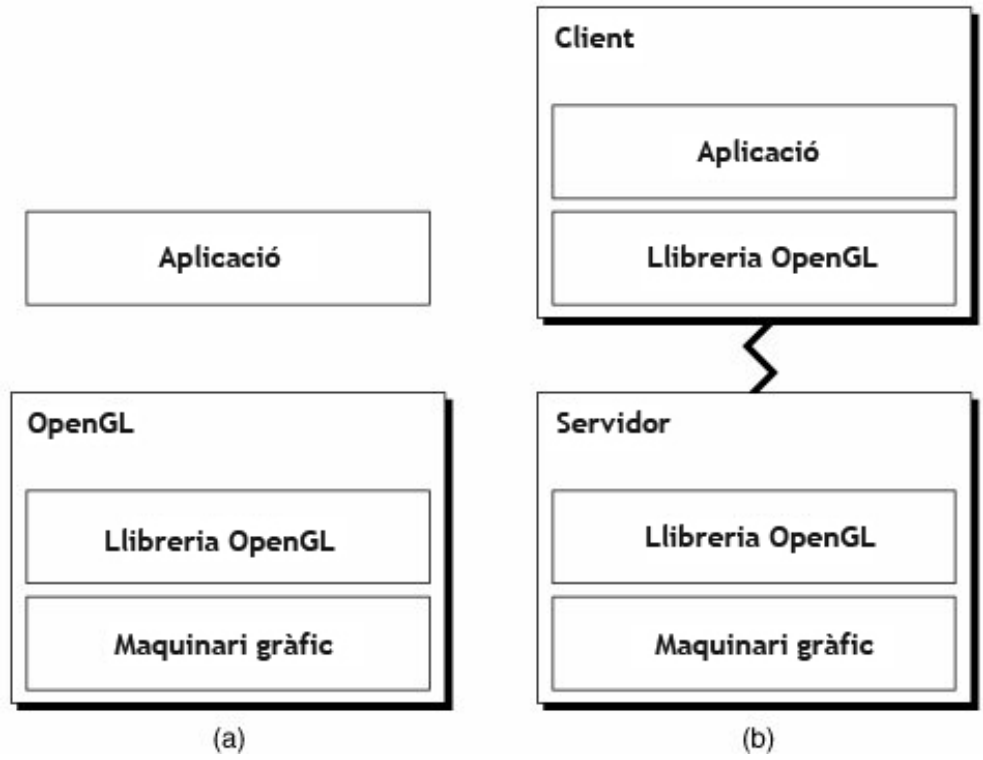


Figura 5: Dues aplicacions típiques *OpenGL*.
(a) Crides des de l'aplicació a la llibreria que interactua amb el maquinari
(b) Estructura client-servidor on les crides passen per la xarxa

De cara al programador, *OpenGL* és un conjunt de comandes que permeten l'especificació d'objectes geomètrics en dues o tres dimensions, juntament amb comandes que controlen com aquests objectes són renderitzats.

La interfície d'*OpenGL* consisteix en un biblioteca de funcions *C/C++*. També existeixen però implementacions diverses que permeten cridar funcions d'*OpenGL* des d'altres llenguatges com ara *Python*, *Perl* i *Java*.

En *OpenGL*, cada nova imatge es redibuixa esborrant el contingut de la finestra i tornant a dibuixar totes les primitives gràfiques de l'escena. Aquest fet és habitual en les aplicacions 3D perquè una petita variació del punt de vista pot fer que canviï pràcticament tot el contingut de la vista.

OpenGL és una màquina d'estats. Les aplicacions criden a les funcions *OpenGL* per establir-ne el seu estat, el qual va determinant l'aspecte final de les primitives en el *framebuffer*.

Un exemple del seu funcionament seria el següent:

S'estableix un estat inicial com per exemple: Color de fons de l'entorn gris, pintat dels polígons per les dues cares, eliminació de la geometria que queda amagada activada i il·luminació habilitada.

```
glClearColor(0.4, 0.4, 0.4, 1.0);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);
```

Ens caldrà definir dues matrius que s'encarregaran de fer la transformació de coordenades de la geometria perquè passin de coordenades de l'aplicació a coordenades de la nostra finestra.

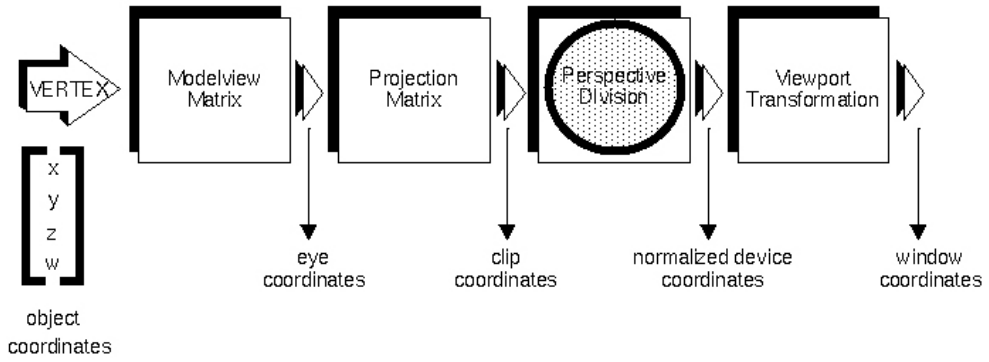


Figura 6: Transformacions de les coordenades de la geometria, d'objecte a finestra

S'inicialitza la matriu que defineix la vista de l'escena. Aquesta matriu anomenada *Modelview* conté uns valors que, al multiplicar-los per les coordenades de la geometria, farà que aquestes es transformin segons la visualització que hem definit.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

S'estableix la informació referent a la il·luminació: components ambient, difusa i especular, i posició de la llum en l'escena.

```
glLightfv(GL_LIGHT1, GL_AMBIENT, llum1Ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, llum1Difus);
glLightfv(GL_LIGHT1, GL_SPECULAR, llum1Especular);
glLightfv(GL_LIGHT1, GL_POSITION, llum1Pos);
glEnable(GL_LIGHT1);
```

Es defineix la càmera des de la que veurem l'escena. S'especifiquen les coordenades de l'observador o càmera (obs), de la posició cap a on mira (vrp) i del vector que determina la inclinació (up). Aquesta instrucció estableix la *Modelview*.

```
gluLookAt(obsX, obsY, obsZ, vrpX, vrpY, vrpZ, upX, upY, _
   _upZ);
```

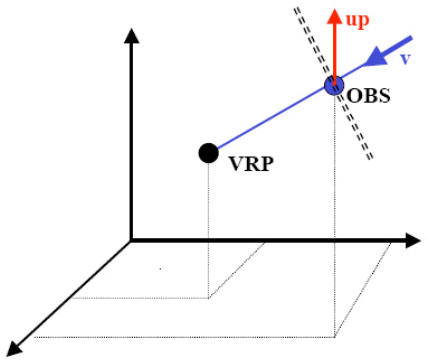


Figura 7: Vectors que determinen la visió de la càmera

Addicionalment s'afegeixen rotacions i/o translacions de la càmera que també modifiquen els valors de la *Modelview*.

```
glTranslatef(vrpX, vrpY, vrpZ);
glRotatef(angleX, 1.0, 0.0, 0.0);
```

```
glRotatef(angleY, 0.0, 1.0, 0.0);
glRotatef(angleZ, 0.0, 0.0, 1.0);
glTranslatef(-vrpX, -vrpY, -vrpZ);
```

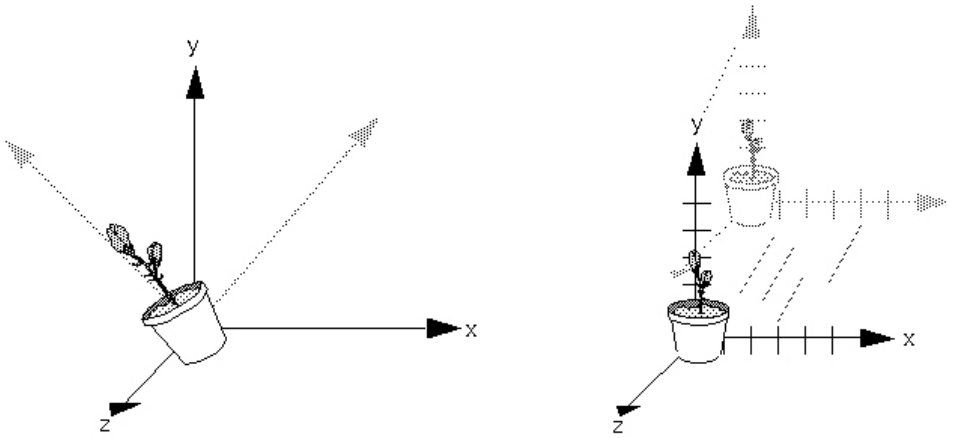


Figura 8: Rotacions i translacions de la càmera

S'inicialitza la matriu que defineix la projecció de la càmera. Es pot definir una visió perspectiva, en forma piramidal amb la punta situada a la càmera, o axonomètrica on la visió té forma de prisma rectangular.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

S'estableix una càmera perspectiva, amb un angle d'obertura, relació d'aspecte i plans de retallat anterior i posterior.

```
gluPerspective(angleCam, width() / height(), near, far);
```

S'estableix el vermell com a color RGB actual.

```
glColor3f(1.0, 0.0, 0.0);
```

S'indica el llançament d'una primitiva. En aquest cas es tracta d'un triangle, així que cada tres punts seran interpretats com a un polígon de tres costats.

```
glBegin(GL_TRIANGLES);
```

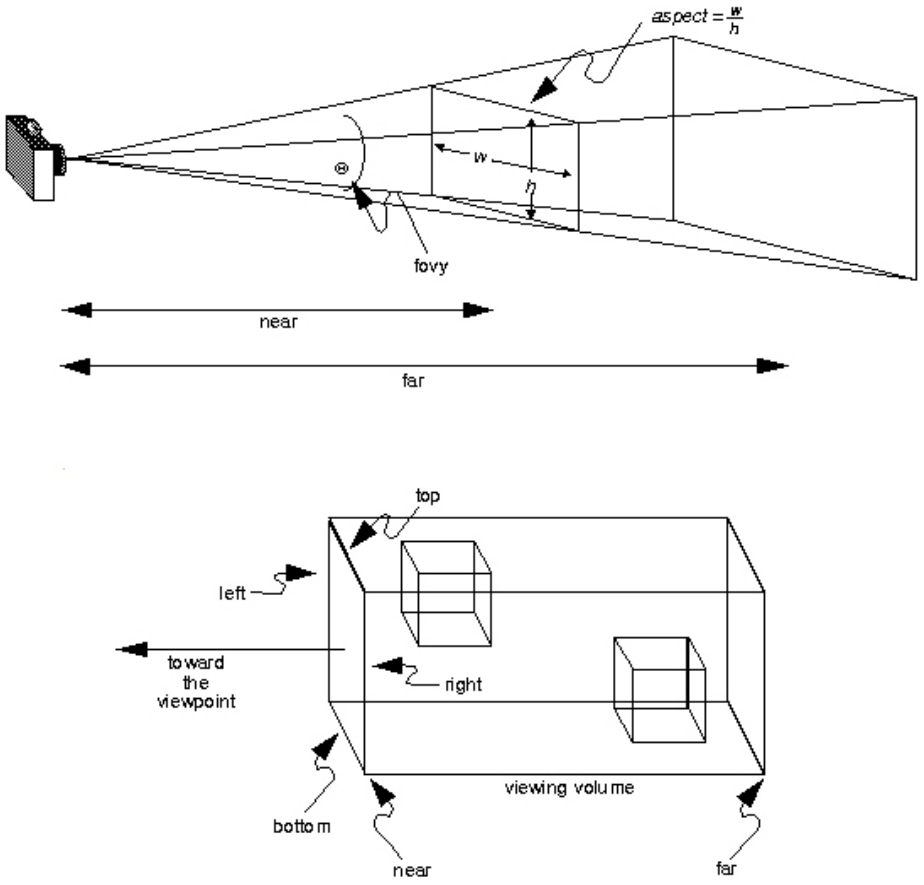


Figura 9: Càmereres perspectiva i axonomètrica

Coordenades x, y i z de cadascun dels tres vèrtexs

```
glVertex3f(-0.5, 0.0, 0.0);
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(0.5, 0.0, 0.0);
```

Hem acabat de llençar la geometria

```
glEnd();
```

La renderització *OpenGL* utilitza una arquitectura *pipeline*, és a dir que les dades van passant una sèrie de fases on se li van aplicant diferents processos fins a obtenir el resultat final.

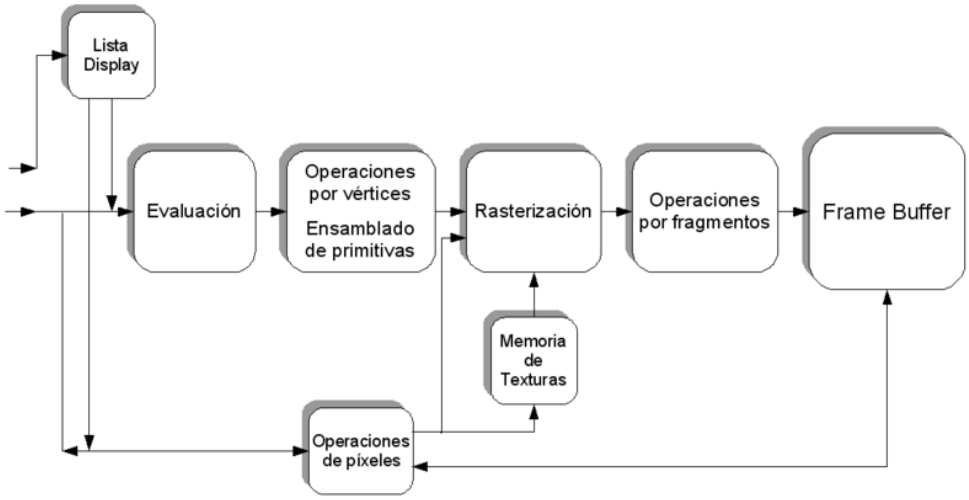


Figura 10: *Pipeline* d'*OpenGL*

1.3.2 Interfícies gràfiques amb *Qt*

Com s'ha comentat anteriorment, una part important de la informàtica gràfica és la utilització d'interfícies de botons per facilitar l'ús dels programes i fer-los més amigables.

En el nostre cas, com que *OpenGL* no proporciona els recursos per crear finestres, hem de recórrer a alguna altra llibreria que ens doni aquesta possibilitat.

Existeixen múltiples llibreries per crear interfícies gràfiques. Podem destacar com a conegudes la llibreria *Swing* de *Java* o les *GTK+* i *Qt* per a *C* i *C++* respectivament.

Figura 11: Logos de Java Swing, GTK+ i Qt

La nostra aplicació està construïda en *C++* i s'ha escollit *Qt* per crear la interfície degut al seu gran ventall de possibilitats i a la bona entesa que té amb *OpenGL*.

Qt (pronunciat *cute*, bonic en anglès) és una llibreria estàndard, desenvolupada per *Trolltech*, molt portable i amb una àmplia llista de funcions per crear tot tipus de finestres, botons i altres *widgets*. La paraula *widget* neix de la fusió de les paraules angleses finestra (*window*) i giny (*gadget*) i es refereix als elements propis de les interfícies gràfiques com barres de desplaçament, botons d'activació, etc.

A part de proporcionar-nos mecanismes per gestionar esdeveniments referents al ratolí i al teclat, *Qt* fa servir un sistema d'emissors i receptors d'accions (*signals* i *slots*) per determinar les conseqüències de prémer un botó, seleccionar un element d'una llista desplegable, etc.

Els *slots* són mètodes especials d'una classe. S'implementen com qualsevol altre mètode però *Qt* els prepara per ser connectats a *signals*. Els *signals* també es declaren com a mètodes però no s'implementen. El funcionament consisteix en indicar en el codi, mitjançant la sentència *emit*, el llançament d'un *signal*, cosa que farà que s'executin el o els *slots* connectats a ell.

Els desenvolupadors de *Qt*, a més, s'han encarregat de crear una aplicació anomenada *Qt Designer* que ens permet construir les interfícies gràfiques d'una manera còmoda i intuïtiva. A base d'anar escollint, col·locant i agrupant *widgets* de tot tipus i definir les connexions entre ells amb el mecanisme de *signals* i *slots*, aviat podem tenir construïdes unes quantes finestres per a la nostra aplicació.

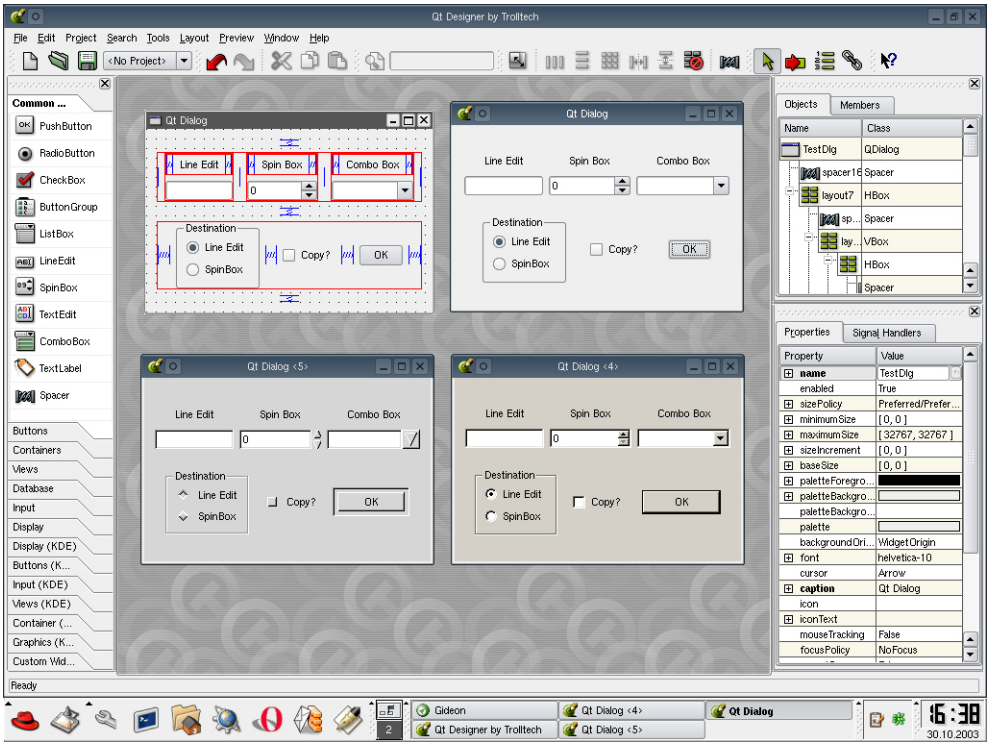


Figura 12: Dissenyador *Qt Designer*

1.3.3 Tractament d'imatges des de *C++*

Un altre dels aspectes importants a tractar en aquest projecte és el processat d'imatges 2D. Un cop més ens trobem amb que hi ha gran quantitat de llibreries dedicades a la gestió d'imatges i un cop més l'elecció havia d'estar en una que ens proporcionés la funcionalitat que necessitàvem i facilitat d'ús. La llibreria escollida ha estat *The CImg Library*.

The CImg Library [14] és un paquet d'eines de codi lliure desenvolupat per David Tschumperlé i altres contribuïdors, que aporta una gran funcionalitat i portabilitat i conté classes construïdes a base de plantilles.

Tot i que la nostra aplicació utilitza només una petita part de la funcionalitat de la llibreria, les seves possibilitats per tractar les imatges píxel a píxel, aplicar-hi filtres i mostrar els resultats, s'ajusta perfectament a les nostres necessitats.

2 Anàlisi de requeriments

El requeriment fonamental del projecte és crear una aplicació que ens permeti fer l'estudi de comparació d'imatges que ens hem plantejat.

A nivell d'aplicació final, el que cal esperar és que el programa ens permeti visualitzar models 3D i enllaçar-los amb imatges obtingudes de la Web.

Del plantejament que s'ha fet de l'aplicació se'n poden derivar els següents requeriments concrets.

2.1 Requeriments funcionals

L'aplicació ha de permetre:

- Visualitzar models 3D.
- Interactuar amb els models. Permetre la visualització des de diferents punts de vista i proximitats.
- Capturar perspectives de l'escena 3D.
- Llegir i mostrar imatges.
- Comparar imatges amb visualitzacions mitjançant diferents algorismes.
- Seleccionar l'algorisme a fer servir.
- Triar si es vol buscar una vista 3D a partir d'una imatge o al revés.
- Estar informat sobre els resultats obtinguts en les comparacions.
- Guardar els resultats obtinguts i recórrer a ells en posteriors visualitzacions.

2.2 Requeriments no funcionals

També és necessari que es compleixin una sèrie de condicions perquè l'aplicació funcioni correctament, sigui usable i puguem extreure conclusions dels resultats.

Requeriments per fer funcionar l'aplicació:

- Es necessita un ordinador amb sistema operatiu *Linux*, llibreries *Qt* i *KDE*, i una versió d'*OpenGL* igual a 1.5 o superior.
- El maquinari gràfic ha de ser prou bo per la feina que ha de fer. La memòria i el processador han d'estar a l'alçada per treballar amb models de gran quantitat de polígons.

Requeriments d'usabilitat:

- S'han de poder llegir els formats dels models 3D amb els que es treballarà, concretament *3DS* i *PLY*.
- L'aplicació ha d'estar desenvolupada amb un llenguatge que proporcioni eficiència al tractar amb gràfics 3D, per exemple *C++*.
- El visor s'ha de comportar de manera fluida per poder-ne fer ús còmodament.
- La interacció amb els models ha de ser intuïtiva.
- La interfície ha de ser clara i simple en concordança amb la senzillesa de la funcionalitat de l'aplicació.

3 Desenvolupament

L'aplicació desenvolupada consta de dues grans parts diferenciades. Per una banda el visualitzador de models 3D i per l'altra els algorismes de comparació d'imatges que analitzaran les imatges obtingudes de la Web i les vistes del model.

3.1 Visualitzador bàsic de models 3D

Per crear el visualitzador 3D amb *OpenGL* s'ha creat una classe amb *C++* per tal de fer les crides necessàries a la llibreria i atendre els esdeveniments de teclat i ratolí.

La classe creada deriva de *QGLWidget*, classe proporcionada per *Qt* que ens permet crear un entorn *OpenGL* dins d'una interfície gràfica feta amb *Qt*.

Tota classe que hereti de *QGLWidget* haurà d'implementar les funcions *initializeGL()*, on s'estableixen variables inicials i es defineixen aspectes que es mantindran constants durant l'execució; *paintGL()*, on es dibuixarà l'escena; i *resizeGL()* on es determina l'àrea ocupada per la vista, on es mostraran les primitives.

A més inclourà la declaració de *signals* i la implementació d'*slots* per poder-se comunicar amb la interfície *Qt*, des d'on es donaran les ordres referents a la comparació d'imatges.

El funcionament de la nostra classe consisteix en llegir el fitxer que conté el model i d'anar pintant a la finestra de visualització els polígons descrits al fitxer segons una sèrie de paràmetres de perspectiva, orientació, etc. determinats per l'estat en el que es trobi *OpenGL*.

Es llegeix també informació referent al color i al material del model, s'esta-

bleixen uns paràmetres d'il·luminació i es dóna l'aspecte final a l'escena 3D.

La finestra de visualització és capaç, a més, de rebre esdeveniments de teclat i de ratolí amb els que ens permetrà rotar el model, apropar-nos-hi i allunyar-nos-en.

3.1.1 Models *3DS*

Existeixen múltiples formats destinats a representar models 3D. Degut a que la funció d'aquesta aplicació no era ni molt menys cobrir un gran ventall d'aquests, s'ha optat per dotar-la de la capacitat de llegir els formats dels models que es tenien a mà durant el desenvolupament.

El primer format tractat és el que correspon a l'extensió ".3ds". Format vinculat al programari propietari *3Dstudio* i destinat a representar malles de triangles amb informació de materials, coordenades de vèrtexs i normals.

Estructura i lectura d'un model *3DS*:

Els models 3DS s'emmagatzemen en una estructura anomenada *Lib3dsFile* i és possible tractar amb ells a través de la llibreria *lib3ds*.

Un *Lib3dsFile* està format per diversos camps amb la informació del model entre els que pararem especial atenció a *Lib3dsMesh* i *Lib3dsNode*.

Un model pot estar format per diversos nodes *Lib3dsNode*, i aquests, relacionats entre ells amb una estructura pare-fill, tenen associades malles de triangles del tipus *Lib3dsMesh*.

Les malles *Lib3dsMesh* es poden obtenir recorrent els nodes del model de manera recursiva i demanant les malles que tenen associades.

```
Lib3dsMesh *mesh = lib3ds_file_mesh_by_name(file3ds, node_
```

```
    _->name);
```

Un cop es té una malla de triangles, es pot accedir a la llista de cares que la conformen.

```
for(Lib3dsDword i = 0; i < mesh->faces; ++i) {
    Lib3dsFace *f = &mesh->faceL[i];
}
```

Finalment de les cares en podem consultar el seus materials i les coordenades dels vèrtexs que les conformen.

```
material = lib3ds_file_material_by_name(file3ds, f->
   _material);
vertex = mesh->pointL[f->points[j]].pos;
```

Aquest procés ens és suficient per interpretar el model i donar instruccions a *OpenGL* perquè ens el pinti.

3.1.2 Models *PLY*

Abançat el projecte, degut a que els models disponibles no eren prou grans comparats amb els que hauria de visualitzar l'aplicació, se'n van buscar de més grossos. Els models que es van poder adquirir estaven representats mitjançant el format lliure *PLY* [13] i així es va compatibilitzar l'aplicació amb aquest format. *PLY* és un format lliure i pensat per ser extensible, és a dir que en cas d'afegir noves propietats als models, aquestes es puguin definir lliurement seguint unes senzilles regles. Inicialment defineix els vèrtexs i les cares del model però ens podem trobar amb fitxers que especifiquin arestes, materials, etc.

Estructura d'un model *PLY*:

Aquest format té dos sub-formats: una representació *ASCII* per a una comprensió i ús més fàcil del model, i una versió binaria per a un emmagatzematge més compacte i més ràpid de guardar i carregar.

El format *PLY* descriu un objecte com una col·lecció de vèrtexs, cares i altres elements, a més de propietats com color i normals que es poden associar a aquests elements. Un fitxer *PLY* conté la descripció d'exactament un objecte.

La definició típica d'un objecte *PLY* és simplement una llista de vèrtexs representats per les seves coordenades x, y i z, i una llista de cares que estan descrites per índexs a la llista de vèrtexs. Vèrtexs i cares són dos exemples dels "elements" que conformen aquests objectes. Cada element té un número fixe de "propietats" que són especificades per a cada element. Les aplicacions poden crear noves propietats associades a elements d'un objecte. Per exemple, les propietats vermell, verd i blau són habitualment associades amb elements del tipus vèrtex. Les noves propietats s'afegeixen de manera que programes que no les comprenguin puguin senzillament no fer-ne cas.

També es poden crear nous tipus d'elements i definir-ne propietats. Exemples d'això seria la definició d'arestes o materials. A l'igual que amb les propietats, els nous elements poden ser omesos per les aplicacions que no els coneixen evitant així que aquestes es trobin en problemes.

L'estructura exacta d'un fitxer *PLY* típic és:

Capçalera
Llista de vèrtexs
Llista de cares
(llistes d'altres elements)

La capçalera és una sèrie de lines de text acabades amb retorn de carro que descriuen la resta del fitxer. Conté una descripció de cada tipus d'element incloent el nom de l'element, quants elements d'aquest tipus conté l'objecte i una llista de les diverses propietats associades a l'element. També

dóna informació referent a si el fitxer és binari o *ASCII*. A continuació de la capçalera hi ha una llista d'elements per cada tipus d'element, presentats en l'ordre descrit a la capçalera.

Exemple d'un fitxer *PLY ASCII* amb la representació d'un cub (les frases entres claus no pertanyen al fitxer, són comentaris per l'exemple):

```
ply
format ascii 1.0 { ascii/binari, versió del format }
comment aqui un comentari { comment introdueix un comentari}
comment aquest fitxer es un cub
element vertex 8 { defineix l'element ‘‘vertex’’, n'hi ha 8 al fitxer }
property float32 x { vertex conté la coordenada ‘‘x’’ de tipus float }
property float32 y { coordenada ‘‘y’’ }
property float32 z { coordenada ‘‘z’’ }
element face 6 { 6 elements del tipus ‘‘face’’ }
property list uint8 int32 vertex_index { definició de llista d'ints }
end_header { final de la capçalera }
0 0 0 { principi de la llista de vèrtexs }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3 { principi de la llista de cares }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

Aquest exemple demostra els components bàsics de la capçalera. Els caràcters "ply<retorn de carro>" han de ser els primers quatre caràcters del fitxer ja que fan la funció de "número màgic". Dóna informació a possibles lectors que es tracta d'un fitxer *PLY*.

3.2 Optimitzacions del pintat de la geometria

Un cop construït el visualitzador bàsic i de cara a donar-li una funcionalitat real dins de l'aplicació ens trobem amb un problema notable. El moviment dels models no resulta gens fluid en el moment en que aquests comencen a tenir un nombre important de triangles, i l'aplicació haurà de tractar amb models de centenars de milers o inclús de milions de polígons.

Així doncs esdevé essencial aplicar optimitzacions que ens donin com a resultat un visualitzador més tractable i apte per a l'aplicació que volem desenvolupar.

Caldrà tenir en compte que, tot i les optimitzacions, la quantitat de càlculs que se li exigeixen a les targetes gràfiques al tractar amb models grossos és molt elevada i per tant segurament es necessitarà un bon maquinari per fer córrer l'aplicació amb comoditat.

El primer pas és detectar la causa de la baixada de rendiment. Si estudiem una mica el comportament d'*OpenGL* veiem que cada vegada que li demanem que ens pinti un vèrtex, les seves coordenades han de viatjar des de la memòria de l'ordinador, on s'hi han guardat momentàniament al llegir-les del fitxer, fins a la memòria de la targeta gràfica perquè el seu processador en pugui disposar. Si a aquest trasllat li afegim el cost de fer una crida a la llibreria i ho multipliquem pels milions de vegades que es requereix aquest procediment cada cop que es vol refrescar la visualització, ens trobem amb el problema mencionat anteriorment.

Els desenvolupadors d'*OpenGL* evidentment han tingut present aquest pro-

blema que és omnipresent en les aplicacions de gràfics per ordinador. Així doncs la llibreria ofereix una sèrie d'opcions per optimitzar tot aquest procés.

El sistema més adient per al nostre visualitzador consisteix en l'empaquetat de la informació de la geometria i l'enviament d'aquesta una única vegada. D'aquesta manera ens estalviem multitud de crides a la llibreria i el que és encara més interessant, la geometria resideix a la memòria de la targeta gràfica i ens podem oblidar del tràfic de dades entre memòria principal i gràfica a cada refresc de l'escena.

3.2.1 *Vertex Arrays* i *Vertex Buffer Objects*

OpenGL utilitza unes estructures de dades anomenades *Vertex Arrays* (d'ara en endavant *VAs*) on s'hi acumula de manera seqüencial tota la informació del mateix tipus. Així doncs per a un model donat, es construiran diversos *VAs*. En el nostre cas un per a les coordenades dels vèrtexs, un altre per a les normals per vèrtex i un tercer per al color dels vèrtexs.

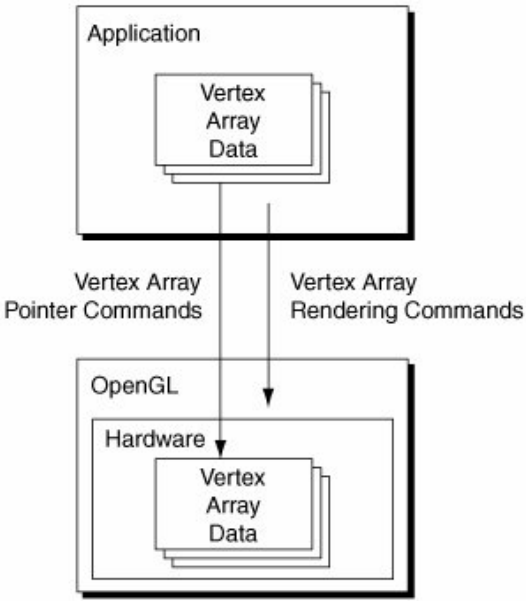


Figura 13: Renderització amb *VAs*, les dades es transfereixen a cada refresc

Els *VAs* per això només cobreixen la part d'empaquetat de la informació. Si ens quedem aquí el moviment de dades en cada refresc ens seguirà fent baixar el rendiment del visualitzador. Amb el temps *OpenGL* va evolucionar la seva optimització mitjançant *VAs* i va crear els *Vertex Buffer Objects* (*VBOs*). Aquests parteixen de la idea anterior però aporten la millora de residència en memòria de la targeta gràfica. Quan creem els *VAs* i fem la crida a la llibreria notificant-li que ha d'interpretar-los coma *VBOs*, *OpenGL* guardarà tota la informació a la targeta i la mantindrà allà per fer-ne ús a cada refresc.

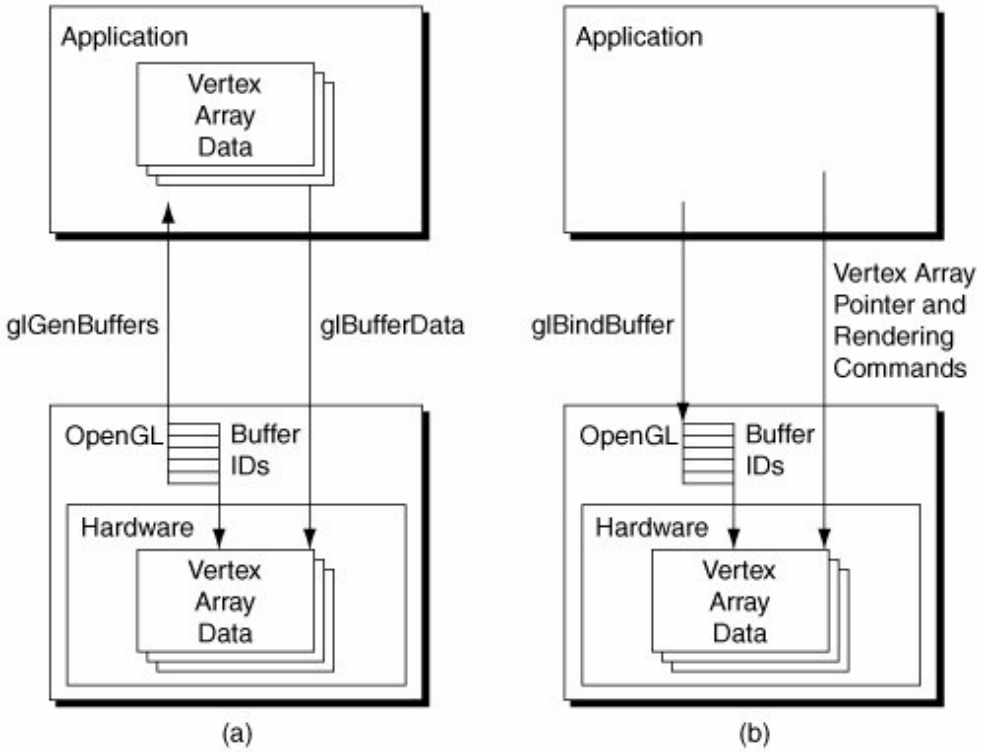


Figura 14: Renderització amb *VBOs*.
(a) S'assigna un ID de buffer i se li assignen les dades.
(b) Al renderitzar s'envien instruccions però les dades ja hi són.

Vegem com s'implementen aquestes millores amb un exemple:

Construïm un vector amb tots els vèrtexs a pintar

```
vertexs[0] = {x0, y0, z0};
vertexs[1] = {x1, y1, z1};
vertexs[2] = {x2, y2, z2};
```

Notifiquem a *OpenGL* que utilitzarem un *VA* de coordenades de vèrtexs

```
glEnableClientState(GL_VERTEX_ARRAY);
```

Si només fem servir *VAs* creem el *Vertex Array* indicant el número de coordenades per vèrtex, el tipus de dades de les coordenades, l'espai entre vèrtexs consecutius dins del vector, i l'adreça de memòria on comença el vector.

```
glVertexPointer(3, GL_FLOAT, 0, &vertexs[0]);
```

Si a més fem servir *VBOs*, hem de crear un vector d'identificadors als *buffers* i indicar quants en farem servir.

```
glGenBuffers(1, &bufObjects);
```

Abans de fer servir un *VBO* s'ha d'activar.

```
glBindBuffer(GL_ARRAY_BUFFER, bufObjects[0]);
```

S'associa el *buffer* actiu amb el *VA* que li vulguem fer correspondre, indicant la mida d'aquest i l'ús que es preveu que es farà de la informació del *buffer*. Depenent de si el contingut es farà servir molt o no i si es modificarà o es mantindrà estàtic, es triarà una opció o una altra. En el nostre cas on la geometria no varia un cop carregat el model, la opció òptima és la de vèrtexs estàtics: GL_STATIC_DRAW.

```
glBufferData(GL_ARRAY_BUFFER, mida, &vertexs[0], _
   _GL_STATIC_DRAW);
```

I finalment creem el *VA* com en el cas anterior però sense determinar una adreça ja que li hem dit a *OpenGL* que faci servir el *VBO* actiu.

```
glVertexPointer(3, GL_FLOAT, 0, 0);
```

Tot aquest procés s'ha de fer una única vegada. Un cop fet això els vèrtexs queden empaquetats i en el cas dels *VBOs* emmagatzemats a la memòria de la targeta gràfica. Cada vegada que es vulguin pintar n'hi haurà prou amb donar-ne l'ordre indicant com s'han d'interpretar els vèrtexs (si com a punts, triangles, quadrats, etc.), l'índex del primer vèrtex a pintar i quants vèrtexs es pintaran.

```
glDrawArrays(GL_TRIANGLES, 0, numTriangles * 3);
```

Hi ha altres funcions per fer el pintat dels vectors com *glDrawElements()* i *glMultiDrawArrays()* que s'hauran de fer servir en cas de voler pintar elements concrets, una secció del vector, etc.

3.2.2 Limitacions

Treballant amb *VAs* i *VBOs* ens trobem un inconvenient que les implementacions d'*OpenGL* actuals encara no tenen resoltes i que té a veure amb la il·luminació de l'escena 3D.

Com s'ha comentat anteriorment, els vèrtexs dels models solen portar associada informació referent al material del que estan fets. Això no és altra cosa que uns paràmetres que especifiquen com li afectarà la llum a aquest vèrtex i per tant quin color adoptarà en una situació d'il·luminació concreta. Aquests paràmetres fan referència a la resposta del material (color que mostrarà) davant de la llum ambient, la difusa i l'especular.

Així doncs quan s'implementa el pintat de la geometria de manera senzilla, per a cada vèrtex es proporcionen aquests tres paràmetres i el nivell de brillantor perquè *OpenGL* sigui capaç d'assignar el color corresponent segons la il·luminació.

Les instruccions per passar aquests paràmetres són *glMaterialfv()* i *glMaterialf()*.

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat->ambient)_
  _;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat->diffuse)_
   _;
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat->_
   _specular);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, phong);
```

A diferència de la informació referent a les coordenades del vèrtex i a les normals, ens trobem amb que no disposem d'un tipus de *VA* destinat a definir materials, només n'hi ha per definir color. Per tant ens veiem obligats a fer servir *shaders* o textures per tal de no perdre informació de color al fer servir *VAs*.

En el nostre cas s'ha utilitzat el color difús de cada vèrtex com a valor vàlid ja que aquest és el que s'acosta més al color final utilitzant materials. Això ho hem fet perquè no ens és tant important que el color mostrat sigui l'exacte i ens podem permetre una petita pèrdua d'informació de color.

3.3 Entorn per la disposició de diverses vistes

Donat que es necessita disposar d'una sèrie de perspectives del model per comparar-les amb les imatges obtingudes de la Web, cal buscar un sistema per distribuir una sèrie de càmeres de la manera més regular possible al seu voltant. A més és interessant que el nombre de càmeres que envolten el model sigui variable ja que segurament models amb més detall necessitaran més perspectives que altres de més senzills.

Una manera molt interessant de crear un entorn regular al voltant de l'escena és construir-hi un icosaedre que l'englobi. El poliedre regular de 20 cares s'utilitza sovint per aproximar una esfera i el centre de cada una de les seves cares serà el lloc ideal per posicionar-hi una càmera. D'aquesta manera ens quedaran 20 càmeres distribuïdes al voltant del model d'una manera molt equilibrada.

En quant a la possibilitat de variar el número de càmeres es pot recór-

rer a un algorisme de subdivisió de políedres. En concret per al nostre cas s'ha utilitzat l'algorisme de *Doo-Sabin*, el qual farà que l'icosaedre inicial s'aproximi més a una esfera cada vegada que s'apliqui l'algorisme al poliedre.

Així doncs utilitzant entre 0 i 2 subdivisions obtenim tres entorns diferents amb 20, 62 i 242 perspectives de càmera diferents respectivament. Aquests entorns seran suficients per als models a visualitzar i estarà a disposició de l'usuari escollir la que més li convingui en cada moment.

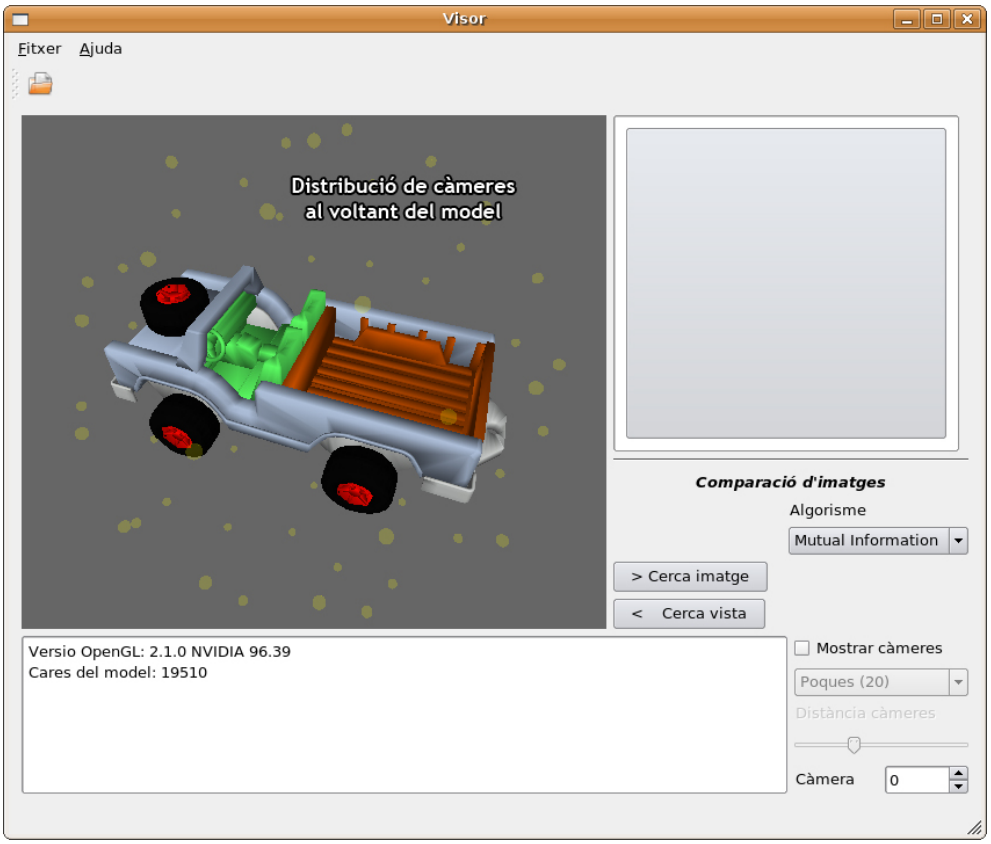


Figura 15: Distribució de càmeres amb l'icosaedre subdividit com a base

3.3.1 Algorisme de Doo-Sabin

Les dues persones que donen nom a l'algorisme van idear aquest mètode de subdivisió de políedres que arrodoneix els objectes geomètrics a base d'afegir cares i reposicionar els vèrtexs.

L'algorisme passa per tres fases diferents.

1. Encongiment de les cares originals
   Cada cara original es redueix de mida sense canviar de posició. Això dóna peu a que es creein noves cares per ocupar els espais buits i les arestes siguin menys anguloses. Els vèrtexs d'una cara doncs s'han de moure dins del pla de la cara de manera que quedin més a prop els uns dels altres. L'elecció de la nova posició dels vèrtexs és fonamental de cara al resultat final. Doo i Sabin van estudiar una sèrie de fórmules per tal de calcular l'encongiment i que l'objecte resultant tingués la mínima energia, és a dir que fos el més arrodonit possible.

   Suposant una cara quadrada amb vèrtexs originals $V1$, $V2$, $V3$ i $V4$, el valor de cadascún dels vèrtexs finals ve descrita per l'expressió

   $$V_2' = \alpha_0 * V_2 + \alpha_1 * V_1 + \alpha_2 * V_3 + \alpha_3 * V_4 \quad (1)$$

   Tenint en compte que el coeficient α_0 correspon al del vèrtex que s'està tractant, les *alphes* s'han de prendre d'acord amb la fórmula

   $$\alpha_0 = 1/4 + \beta_0 \alpha_i = \beta_i \quad (2)$$

   amb i de 1 a $K - 1$, sent K el número de vèrtexs de la cara, i β es defineix com

   $$\beta_j = 1/4(3 + 2 * cos(2 * \pi/K)) * 1/K \quad (3)$$

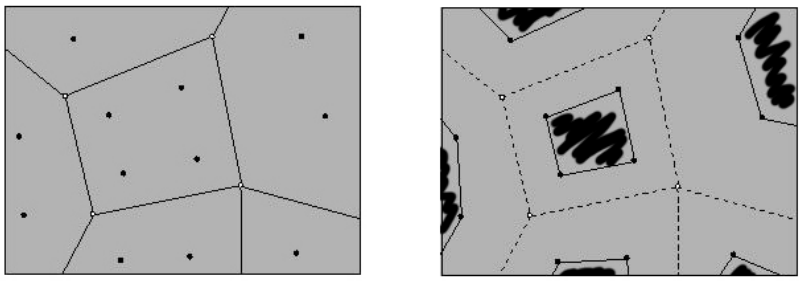


Figura 16: Encongiment de les cares originals

2. Noves cares per cada antiga aresta
   Per començar a omplir els forats que ha deixat el procés d'encongiment de cares, es crea una cara quadrada per cada antiga aresta. Les arestes anteriors s'han desdoblat i separat, les noves cares doncs enllacen les dues arestes que antigament eren la mateixa.

3. Noves cares per cada vèrtex antic
   Únicament queda per omplir l'espai on se situaven els antics vèrtex. En aquest cas s'enllacen les cares que abans tenien un mateix vèrtex comú formant noves cares de tants costats com cares antigues compartien el vèrtex.

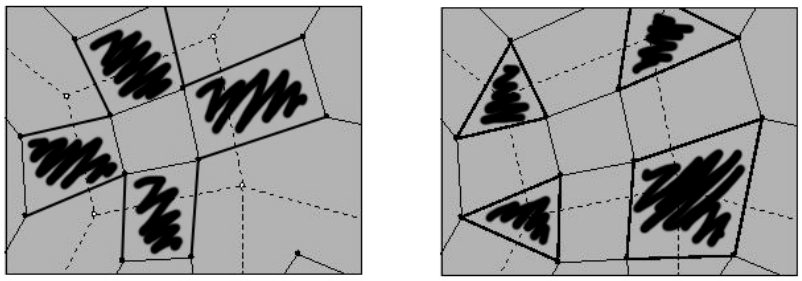


Figura 17: Noves cares de les antigues arestes i antics vèrtexs

Després d'aquest procés, l'objecte geomètric ha multiplicat el seu nombre de cares i ha adquirit un aspecte molt més arrodonit que abans. L'algorisme

es pot aplicar una vegada i una altra sobre els models resultants suavitzant així cada vegada més l'objecte.

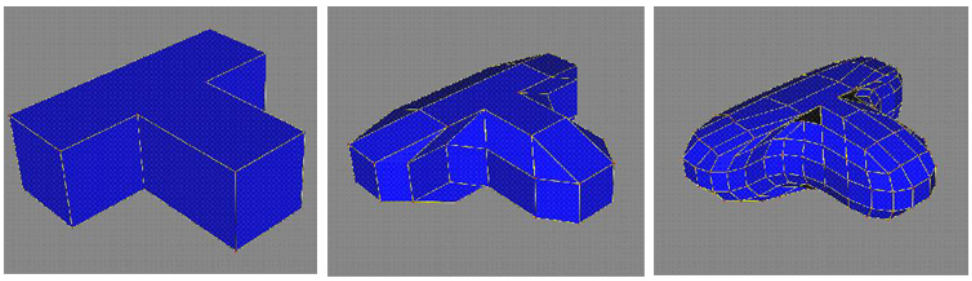


Figura 18: Poliedre subdividit amb Doo-Sabin

3.4 Algorismes de comparació d'imatges

La segona part important d'aquest projecte i la que dóna la utilitat final és la que correspon a comparar les imatges de la Web amb perspectives dels models 3D.

Per fer això s'han estudiat una sèrie d'algorismes de comparació d'imatges basats en diversos conceptes i s'ha creat una classe amb la implementació d'alguns d'ells per a l'aplicació.

El procés per fer les comparacions és el següent:

- Es captura la perspectiva actual de l'escena 3D utilitzant la funció *glReadPixels()* d'*OpenGL* que ens dóna el valor de cada píxel del visor.
- Amb la llibreria *CImg* convertim la captura i la imatge amb la que l'hem de comparar en objectes imatge.
- Es crida a l'algorisme escollit amb les dues imatges.
- Si l'algorisme ho requereix es redimensionen les imatges perquè tinguin la mateixa mida.

- Finalment es retorna el valor calculat per l'algorisme i es compara amb els altres resultats obtinguts.

3.4.1 *Mutual Information*

Els mètodes basats en la informació mútua (*Mutual Information*) formen un grup aparegut recentment i representen la tècnica capdavantera en l'anàlisi d'imatges multimodals.

La *Mutual Information* és una mesura de dependència estadística entre dos variables aleatòries. Es basa en l'entropia de les dades, és a dir la quantitat d'informació que contenen. En el cas de les imatges, una imatge amb molt detall tindrà més informació que una imatge més llisa. L'aplicació de la informació mútua és particularment adequada per l'anàlisi d'imatges capturades mitjançant diferents modalitats. Un exemple seria la comparació d'una fotografia feta amb raigs X i una altra en la que s'han utilitzat ultrasons. Degut a que situacions com aquesta es produeixen sovint a la realitat, el mètodes basats en la informació mútua resulten molt interessants en el camp de la medicina.

La informació mútua entre dues imatges X i Y ve donada per:

$$MI(X,Y) = H(X) + H(Y) - H(X,Y) \tag{4}$$

on $H(X)$ és l'entropia de la imatge X i $H(X,Y)$ és l'entropia conjunta de X i Y.
L'entropia ve donada per l'expressió:

$$H(X) = -\sum_{i=0}^{n} p_i * \log_2 p_i \tag{5}$$

i p_i representa el nombre de píxels de la imatge amb valor i dividit pel nombre total de píxels de la imatge.

L'entropia conjunta s'expressa:

$$H(X,Y) = -\sum_{i=0}^{n} p_{ij} * \log_2 p_{ij} \quad (6)$$

i p_{ij} representa el nombre de vegades en que un píxel corresponent a la imatge X pren valor i, i el mateix píxel de la imatge Y pren el valor j; dividit pel nombre total de píxels de la imatge (cal tenir en compte que les dues imatges han de tenir el mateix nombre de píxels).
Intuïtivament, la informació mútua mesura la informació que comparteixen X i Y, en quina proporció conèixer una de les variables redueix la incertesa sobre l'altra. Per exemple, si les variables són independents, el coneixement d'una no dóna cap informació sobre l'altra i per tant la seva informació mútua és zero. Per altra banda si X i Y són idèntiques, aleshores tota la informació de X és compartida per Y, conèixer una et dóna el valor de l'altra.
Aquesta capacitat d'avaluar la diferència d'informació entre les dues imatges és el que li atorga l'habilitat per detectar característiques en les imatges, cosa que la fa útil per la comparació entre imatges de fonts diferents.

3.4.2 *Normalized Compression Distance*

Si dos fitxers, per exemple, són similars d'acord amb una característica particular definida per una mètrica concreta, també ho són en la mètrica d'informació normalitzada. Aquesta mètrica està basada en la noció de complexitat de *Kolmogorov*. La complexitat de *Kolmogorov* d'un fitxer és en essència la longitud de la versió comprimida definitiva del fitxer. Desafortunadament la complexitat de *Kolmogorov* d'un fitxer no és computable, caldrà doncs fer servir compressors que s'aproximin al màxim a la perfecció requerida. L'aproximació resultant de la "normalized information distance" s'anomena *Normalized Compression Distance (NCD)* i esdevé una mètrica de similitud.

La *NCD* pren com a arguments dos objectes i avalua una fórmula expressada en termes de la versió comprimida d'aquests objectes, separadament i de forma combinada. D'aquesta manera, està parametritzada pel compres-

sor utilitzat. Si X i Y són els dos objectes implicats i $C(X)$ és la longitud de la versió comprimida de X utilitzat el compressor C, aleshores la *NCD* de X i Y:

$$NCD(X,Y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \quad (7)$$

En el cas que ens ocupa, els dos objectes involucrats en la fórmula seran les dues imatges a comparar. Si les dues imatges són similars en quant a quantitat d'informació (quant menys plana és una imatge, més informació té), la versió comprimida d'un tindrà pràcticament la mateixa mida que la versió comprimida de l'altre.

La *NCD* és un valor no negatiu r tal que $0 <= r <= 1 + \epsilon$ que representa què tant diferents són els dos arxius. Quant més petit sigui el valor, més semblants són. La constant ϵ es deguda a imperfeccions en el compressor però pels algorismes de compressió més estàndards com *gzip*, *bzip2* o *PPMZ*, és difícil trobar valors per sobre del 0,1. El nostre algorisme utilitza *bzip2* per construir les versions comprimides de les imatges.

3.4.3 *Scale-Invariant Feature Transform*

Scale-Invariant feature transform (*SIFT*) és un algorisme per detectar i descriure característiques locals en imatges. La detecció de característiques pot ajudar a reconèixer objectes i no es veu afectada per rotacions o escalats. També és robusta a diferències d'il·luminació, soroll i en certa mesura a oclusions.

L'algorisme passa per diverses fases en les que detecta els punts claus, els avalua i finalment decideix quines són les interessants. Aquests punts claus són els que després s'utilitzen per relacionar les imatges entre si.

Aquest algorisme no s'ha implementat ja que presentava un grau de complexitat molt elevat i no oferia masses garanties d'èxit. S'utilitza sobretot per reconèixer característiques iguals entre dues imatges coneixent que

existeixen. En el nostre cas en el que es comparen imatges que poden no tenir res a veure i justament cal dir es corresponen o no, podria no aportar resultats massa útils.

Figura 19: Imatges a relacionar

Figura 20: Enllaç de punts claus entre les imatges

3.5 Etiquetat dels models

La idea plantejada inicialment era que les comparacions entre models i imatges donessin com a resultat uns vincles que quedarien emmagatzemats. Així un model al qual se li havien assignat una sèrie d'imatges fruit de les comparacions, quedaria enriquit amb aquestes imatges. Les següents vegades que es visualitzés el model, es mostrarien, juntament amb ell, aquestes imatges assignades.

Finalment la feina s'ha centrat més en l'estudi de les optimitzacions i l'aplicabilitat dels algorismes de comparació i ha deixat una mica de banda aquest primer plantejament.

De tota manera s'ha volgut tenir en consideració la idea de guardar els resultats de les comparacions i el que s'ha fet finalment és mantenir un registre de la comparació més satisfactòria per a cada model.

Al obrir un model es crea un arxiu de configuració, si no el té ja, destinat exclusivament a ell. Aquest arxiu porta el nom del model amb extensió ".mtx" i es guarda a la carpeta "configs" de l'aplicació. Un cop creat, l'arxiu informa de que encara no s'ha assignat cap imatge al model.

Després de la primera comparació, aquesta quedarà registrada en el fitxer de configuració, on hi constarà el nom de la imatge relacionada, l'algorisme utilitzat i el valor obtingut en la comparació. En les conseqüents comparacions, es mirarà si el resultat obtingut és millor que el registrat en l'arxiu de configuració, i de ser així es reemplaçarà la informació. S'aconsegueix així que amb el temps els models tinguin associada una imatge que es correspon fortament amb ells.

Al quedar etiquetats d'aquesta manera, cada vegada que s'obre un model es veu amb ell la imatge associada ja que es consulta de forma automàtica el seu fitxer de configuració i es carrega la imatge pertinent.

4 Conclusions

4.1 Resultats de l'aplicació d'optimitzacions

Per tal de poder fer una valoració de la millora de rendiment obtinguda en l'aplicació d'optimitzacions, durant el desenvolupament s'han mantingut operatius els diferents mètodes de pintat de la geometria. És a dir, partint del mètode senzill que utilitza les instruccions *glBegin() / glEnd()*, s'ha implementat l'ús de *VAs* i posteriorment de *VBOs* donant opció de passar d'un sistema a l'altre en qualsevol moment.

Així com a primera vista, amb models petits, no hi ha cap diferència entre els tres mètodes, en el moment en que es visualitzen models amb un nombre de polígons més elevat la millora comença a notar-se. Mentre que el moviment ràpid del model comença a perdre fluïdesa amb el primer mètode, amb l'ús de *VAs* i *VBOs* seguim obtenint un rendiment excel·lent.

Per poder demostrar l'augment de rendiment que hem observat, s'han portat a terme una sèrie de proves per obtenir freqüències de mostreig i poder comparar numèricament el rendiment dels diferents mètodes. Ens interessa especialment poder comprovar les diferències entre l'ús de *VAs* i *VBOs* ja que a simple vista no són evidents.

Les proves han consistit en fer rotar els models sobre ells mateixos durant deu segons i comptar la quantitat de vegades que es dibuixa la geometria en pantalla. S'han fet servir tres models de mides diferents, un petit, un segon amb un nombre de triangles més elevat i un últim considerablement gran.

El maquinari utilitzat per realitzar les proves ha estat el següent:

Processador	AMD Athlon 64 bits 3500+ 2,21 GHz
Memòria RAM	1 GB
Targeta gràfica	NVIDIA GeForce 7300 SE 64MB PCI Express

Taula 1: Maquinari gràfic

Vegem-ne els resultats:

Nom model: dolphin.3ds
Número de cares: 4.422

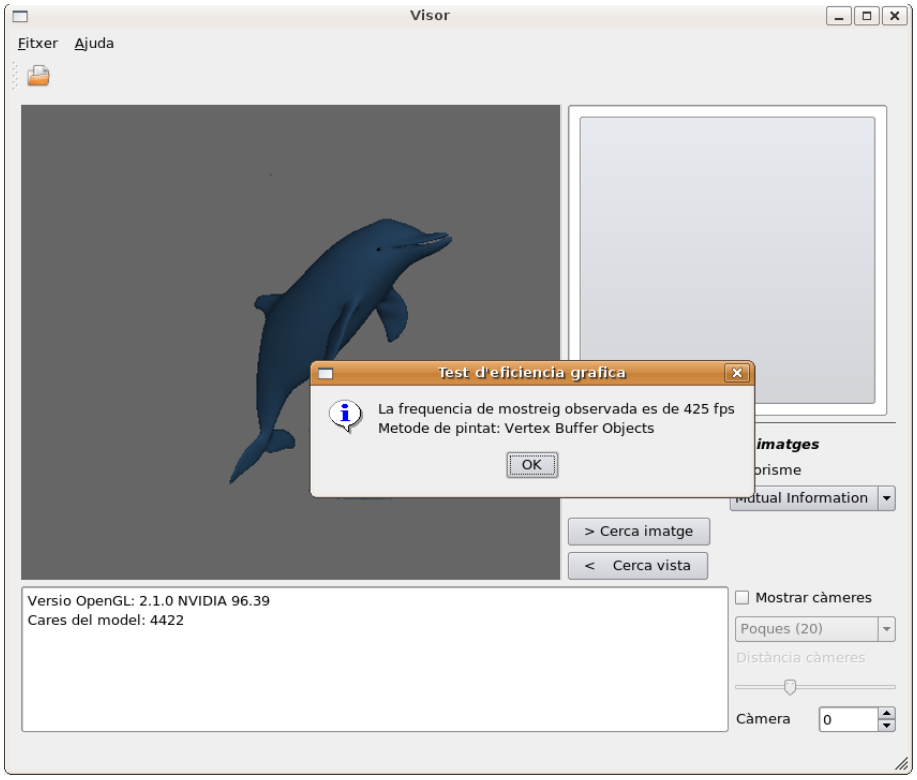


Figura 21: Rendiment amb un model petit

Freqüències de mostreig:

Sense optimitzacions	**VAs**	**VBOs**
203	399	406
194	408	417
195	399	423
170	419	404
199	415	431
186	434	437
200	413	398
193	401	412
200	420	411
200	399	406
194	**410,7**	**414,5**

Taula 2: Freqüències de mostreig per a dolphin.3ds

Gràfiques de rendiment:

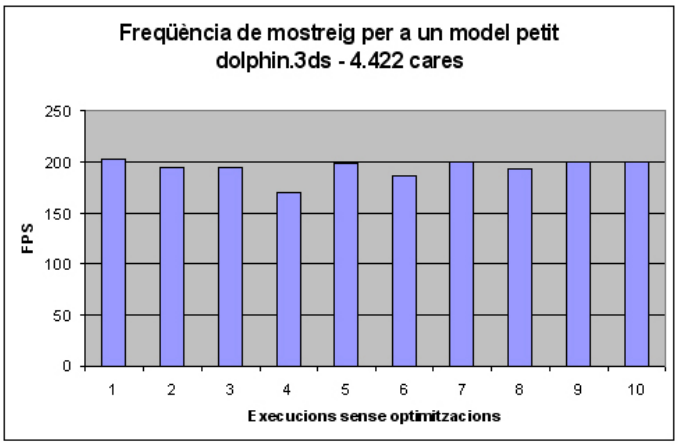


Figura 22: Rendiment sense optimitzacions per a dolphin.3ds

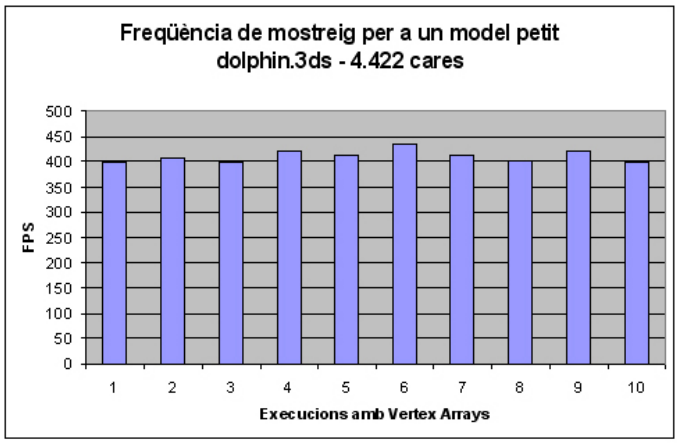


Figura 23: Rendiment amb *VAs* per a dolphin.3ds

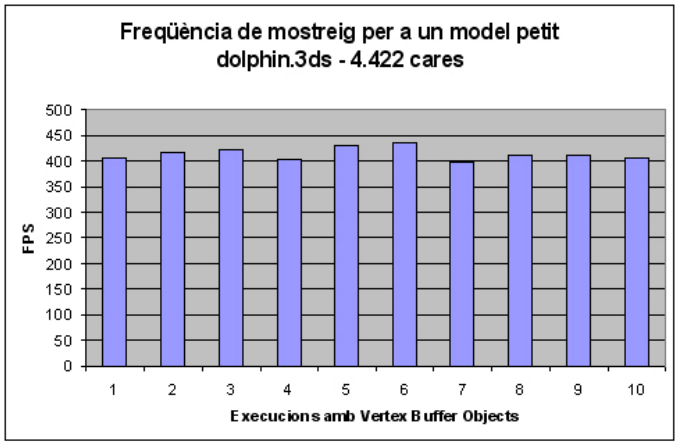


Figura 24: Rendiment amb *VBOs* per a dolphin.3ds

Comparació entre els mètodes de pintat:

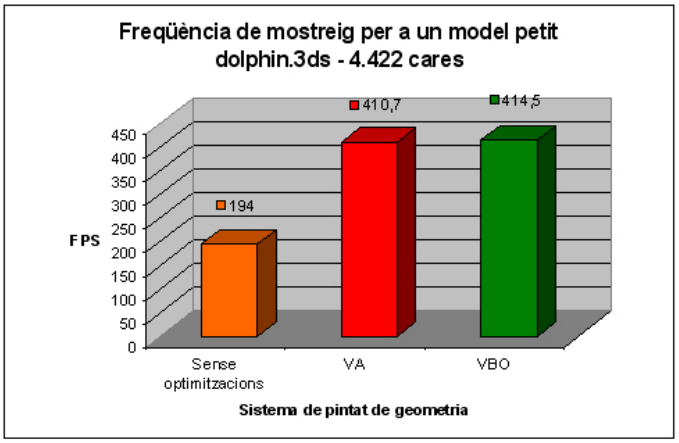


Figura 25: Comparació de rendiments per a dolphin.3ds

En aquest últim gràfic podem apreciar que les optimitzacions aporten més del doble de rendiment respecte al pintat senzill. També veiem que la diferència entre *VAs* i *VBOs* no és massa notable per a models petits.

Nom model: executor.3ds
Número de cares: 75.818

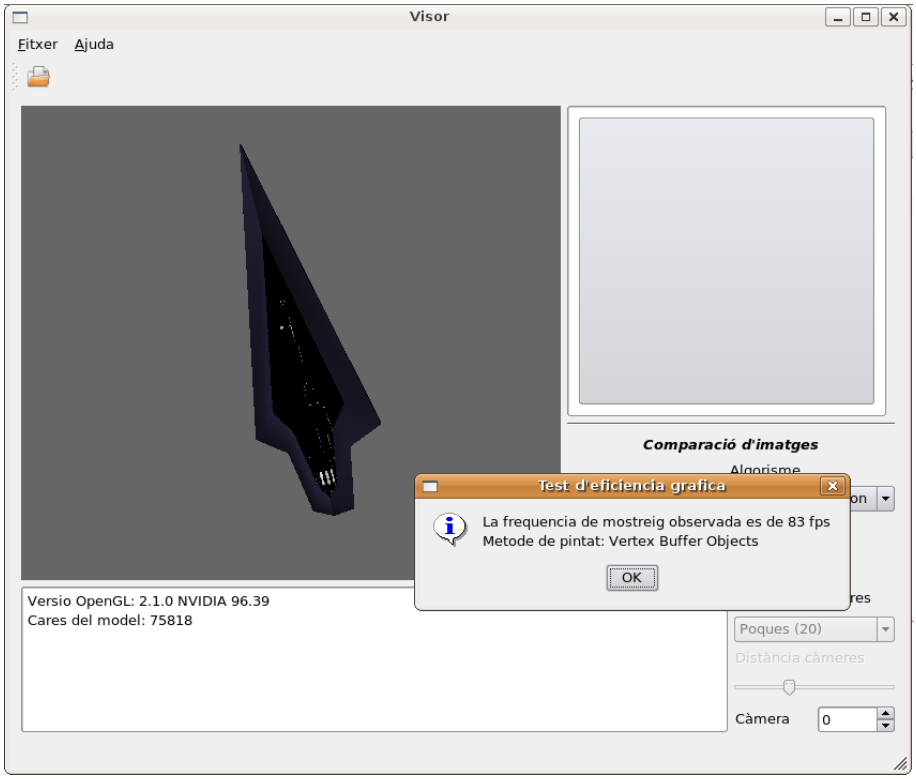


Figura 26: Rendiment amb un model mitjà

Freqüències de mostreig:

Sense optimitzacions	**VAs**	**VBOs**
10	96	102
11	102	96
11	98	94
11	103	97
11	96	97
11	103	94
12	100	103
11	96	100
12	96	99
10	98	97
11	**98,8**	**97,9**

Taula 3: Freqüències de mostreig per a executor.3ds

Gràfiques de rendiment:

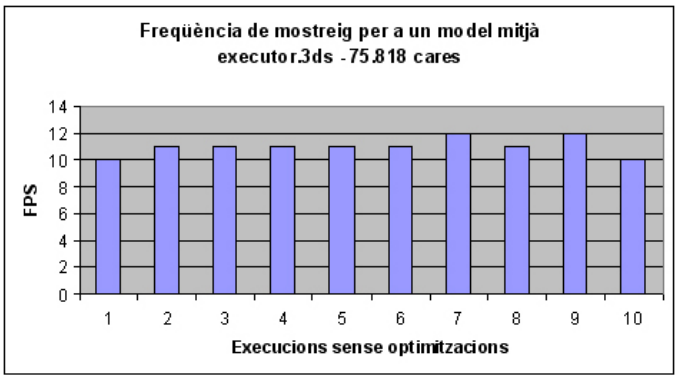


Figura 27: Rendiment sense optimitzacions per a executor.3ds

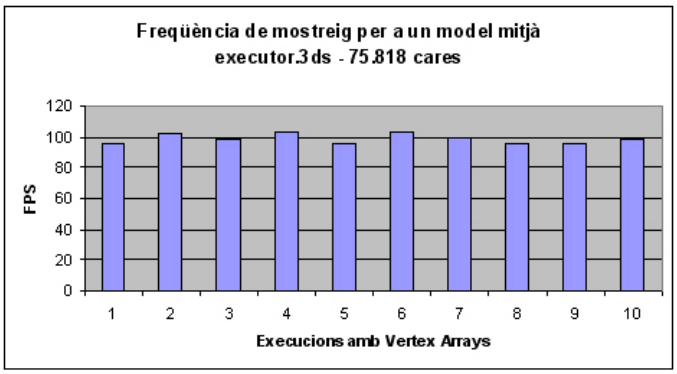


Figura 28: Rendiment amb *VAs* per a executor.3ds

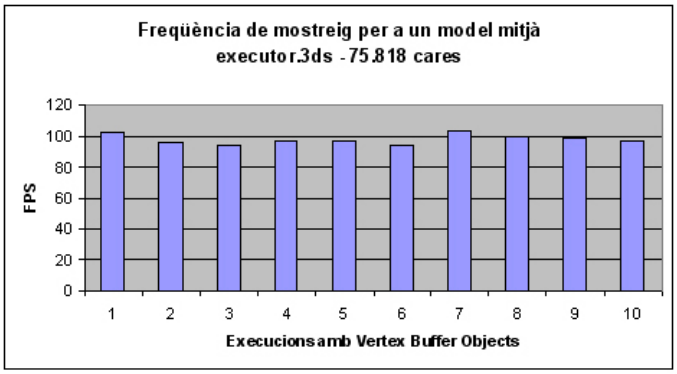


Figura 29: Rendiment amb *VBOs* per a executor.3ds

Comparació entre els mètodes de pintat:

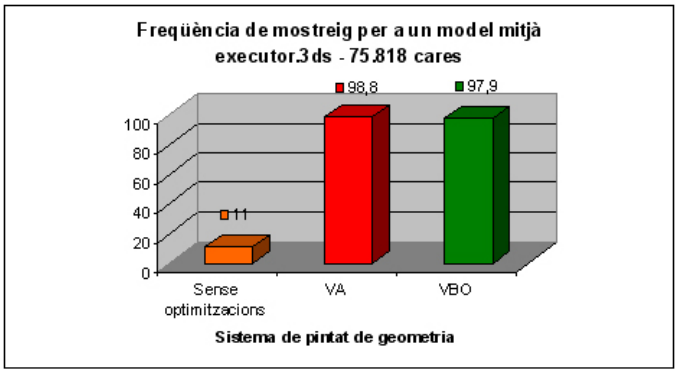


Figura 30: Comparació de rendiments per a executor.3ds

Aquí ja es veu que l'ús de les optimitzacions aporta una millora molt important. En aquest cas s'ha multiplicat el rendiment per deu.

Nom model: dragon.ply
Número de cares: 871.414

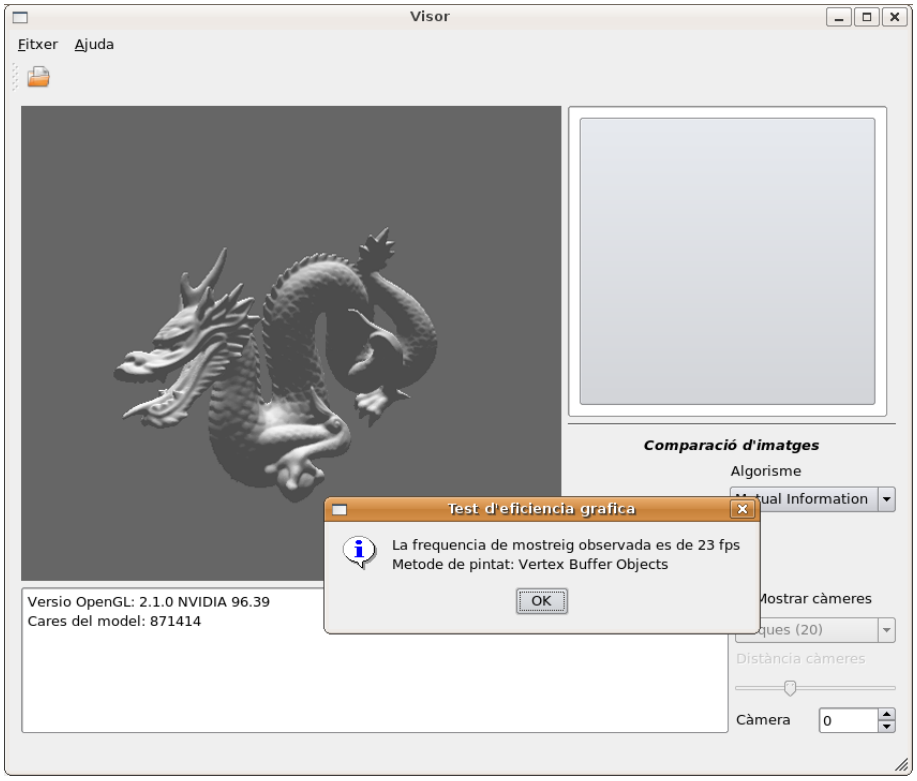


Figura 31: Rendiment amb un model gran

Freqüències de mostreig:

Sense optimitzacions	**VAs**	**VBOs**
0	20	21
0	20	21
0	20	21
0	21	21
0	21	21
0	**20,4**	**21**

Taula 4: Freqüències de mostreig per a dragon.ply

Gràfiques de rendiment:

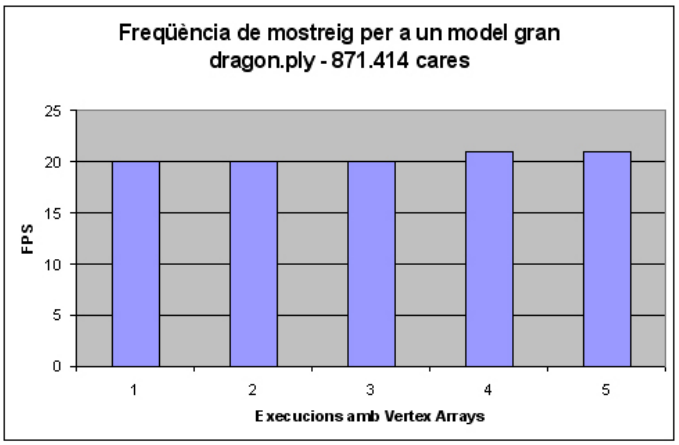


Figura 32: Rendiment amb *VAs* per a dragon.ply

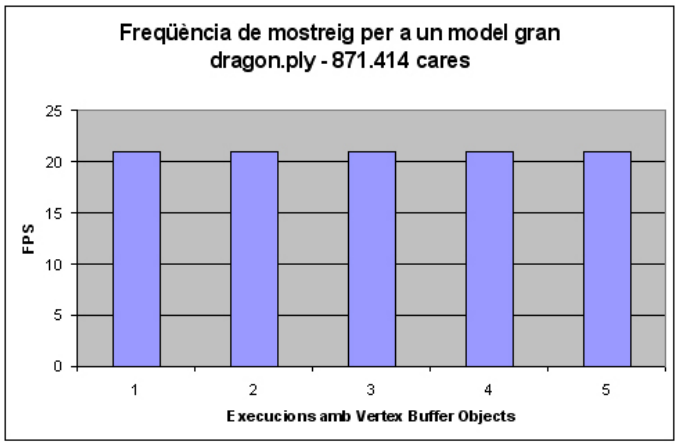


Figura 33: Rendiment amb *VBOs* per a dragon.ply

Comparació entre els mètodes de pintat:

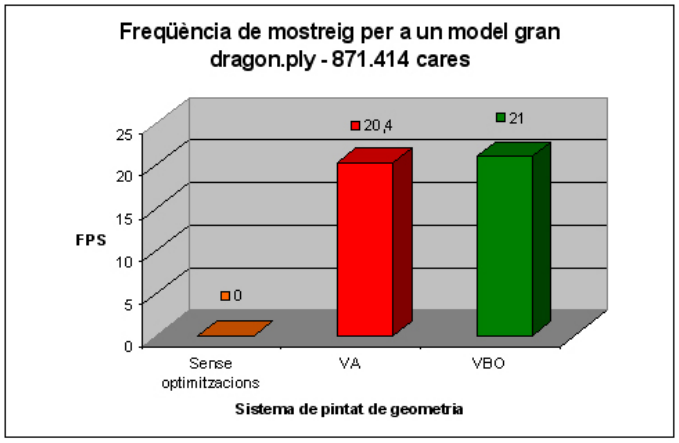


Figura 34: Comparació de rendiments per a dragon.ply

Sense optimitzacions, models d'aquestes dimensions ja resulten immanejables, fins al punt d'arribar a bloquejar l'aplicació.

Tant en el model mitjà com en el gran hem pogut observar que l'ús de *VBOs* no suposa una millora substancial respecte als *VAs*, inclús alguns resultats obtinguts són pitjors. Per altra banda, pel que hem estudiat, sembla que haurien d'aportar alguna millora. La deducció que en traiem és que la

seva aplicació farà millorar el rendiment en situacions concretes en les que s'exprimeixi la seva capacitat de retenir les dades en memòria de la targeta gràfica. Podem afegir també que tot i no haver-ho pogut comprovar, un motiu d'aquest comportament podria ser degut a que la targeta no té prou memòria per emmagatzemar tota la geometria i això anés en detriment del rendiment. S'explicaria així que pel model mitjà els *VBOs* donin millors resultats que en el model gros.

Evidentment el punt en que les optimitzacions aplicades aquí resulten suficients per la correcta visualització de l'escena és real i si s'ha de desenvolupar una aplicació que es troba amb aquest problema caldrà estudiar altres millores existents i aplicar les més adients per la necessitat que hi hagi.

Com s'ha comentat anteriorment, la utilització d'aquests sistemes d'optimització ens ha aportat un problema referent a la il·luminació que no s'havia previst. No és possible construir *VAs* amb informació de materials, cosa que ens fa perdre informació referent al color dels objectes il·luminats.

Un recurs per solucionar aquest problema és utilitzar *shaders*. El sistema de *VAs* permet enviar informació perquè sigui tractada dins la targeta i és aquí on s'hauria de fer el càlcul pertinent per donar el color autèntic a la geometria.

En el cas de disposar d'elles també podríem fer ús de textures que li donessin l'aspecte desitjat al model.

Per el nostre cas concret podem concloure que les optimitzacions ens han aportat una millora de rendiment més que notable i amb elles hem assolit l'objectiu plantejat de mostrar una escena 3D correcta i interactuable.

4.2 Resultats dels algorismes de comparació

Finalment els algorismes implementats d'entre els que s'han vist han estat el de *Mutual Information* i el de la *Normalized Compression Distance*. Un cop implementats, cal veure si ens aporten la funcionalitat que es busca.

Per verificar la seva efectivitat s'han realitzat unes quants proves de comparació entre models i imatges escollides prèviament. A part d'algunes imatges concretes que corresponen directament al model i ens han permès fer una valoració més específica dels resultats, la majoria són imatges obtingudes amb el buscador Google Imatges. D'aquesta manera hem pogut avaluar el comportament en una situació realista.

El que s'ha observat d'entrada, tot i ser previsible, és que els costos d'aplicar els algorismes és elevat i execucions amb grans volums de dades podrien allargar-se notablement.

Les imatges utilitzades per la bateria de proves són aquestes: Les imatges

Figura 35: Imatges per la bateria de proves

1, 2 i 3 es corresponen amb el model 3D *Happy Buddha* i les farem servir per comprovar l'efectivitat dels algorismes en condicions òptimes. Les 7, 8, 12 i de la 13 a la 19 són imatges de la mateix figura però en aquest cas són fotografies reals i algunes tractades digitalment. I la resta, amb excepció del cap del dofí, són fotos de budes diferents al del model.

Vegem doncs els resultats de les proves:

Primera prova
Amb una visió frontal del buda, cerca la imatge més similar.
Resultat esperat: El valor més alt hauria de correspondre a la imatge 1 i s'haurien d'obtenir bons resultats per a les imatges 8, 12, 13 - 19.

Resultats per a l'algorisme de *Mutual Information*:

Número imatge	**Valor (quant més gran millor)**
1	**0,406316**
2	0,358555
3	0,276008
4	0,649085
5	0,263303
6	0,338328
7	0,387123
8	0,371651
9	0,366647
10	0,405485
11	0,201577
12	0,371214
13	0,055661
14	0,123611
15	0,108862
16	0,088399
17	0,160388
18	0,079803
19	0,136475

Taula 5: Resultats de *MI* amb el model en posició frontal

Hem obtingut com a imatge resultant la que esperàvem, la primera. Les imatges 8 i 12 es troben entre les vuit millors, però cap fotografia entre la 13 i la 19 han obtingut gaires bons resultats. També resulten sorprenents que hagin quedat per sobre imatges com la 6 o la 10.

És difícil deduir el motiu d'aquesta poca efectivitat, possiblement vagi associat al redimensionament que han de patir les imatges per poder-se comparar píxel a píxel o a la incapacitat de l'algorisme per detectar fronteres. Ambdues coses són material per a un possible treball futur.

Resultats per a l'algorisme de *Normalized Compression Distance*:

Número imatge	**Valor (quant més petit millor)**
1	1,01114
2	**0,99859**
3	1,01194
4	1,03087
5	1,01069
6	1,00376
7	1,01901
8	1,01839
9	1,03553
10	1,02715
11	1,02451
12	1,04319
13	1,03783
14	1,02469
15	1,04026
16	1,04495
17	1,03005
18	1,04456
19	1,02848

Taula 6: Resultats de *NCD* amb el model en posició frontal

En aquest cas la imatge que esperàvem obtenir ha quedat en tercer lloc precedida per les altres dues perspectives del model, les imatges 2 i 3.

Les imatges 8 i 12 han obtingut bons resultats però no tant les últimes. El fet de contenir gran quantitat d'ombres i reflexos pot produir que la

quantitat de colors augmenti i per tant la mida de la imatge.

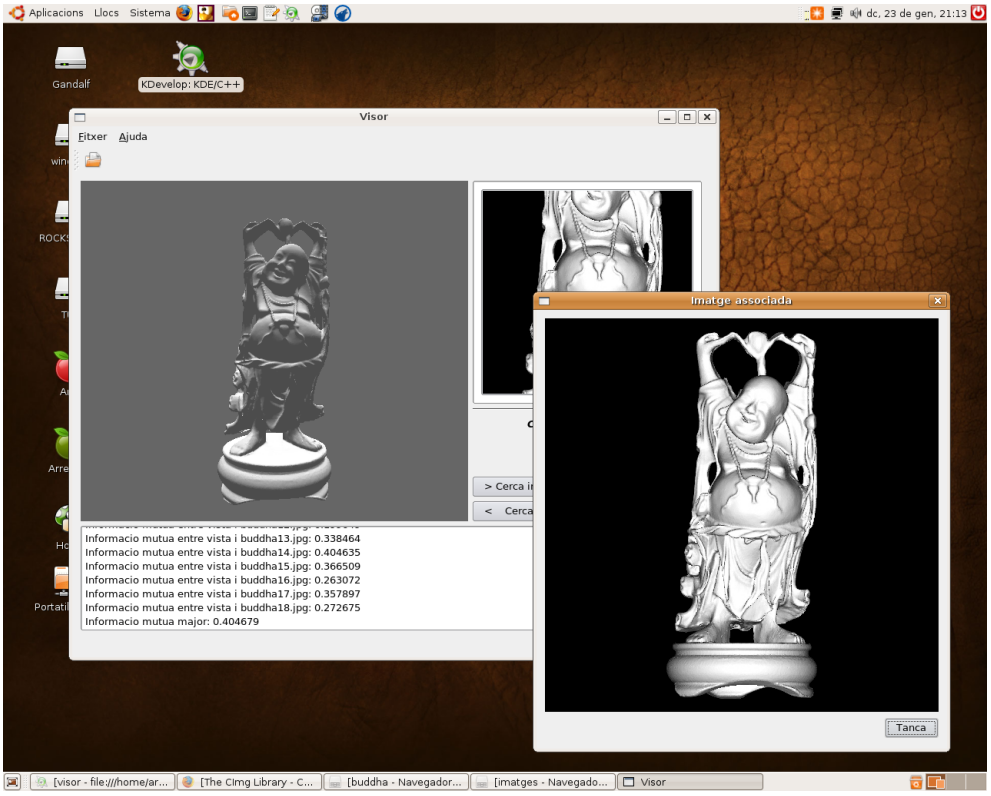


Figura 36: Resultat de la cerca d'imatges feta per *MI* amb el model en posició frontal

Segona prova
Visió lateral del buda, cerca de la imatge més similar.
Resultat esperat: La imatge número 2 hauria de ser l'escollida.

Resultats per a l'algorisme de *Mutual Information*:

Número imatge	**Valor (quant més gran millor)**
2	0,38817
10	**0,40877**

Taula 7: Resultats de *MI* amb el model en posició lateral

Resultats per a l'algorisme de *Normalized Compression Distance*:

Número imatge	**Valor (quant més petit millor)**
2	1,00879
5	**1,00595**

Taula 8: Resultats de *NCD* amb el model en posició lateral

Els dos algorismes donen la imatge 2 com a segona opció mentre que la *MI* escolleix la 10 com a millor i la *NCD* tria la 5. En aquest cas els resultats són potser una mica més inesperats.

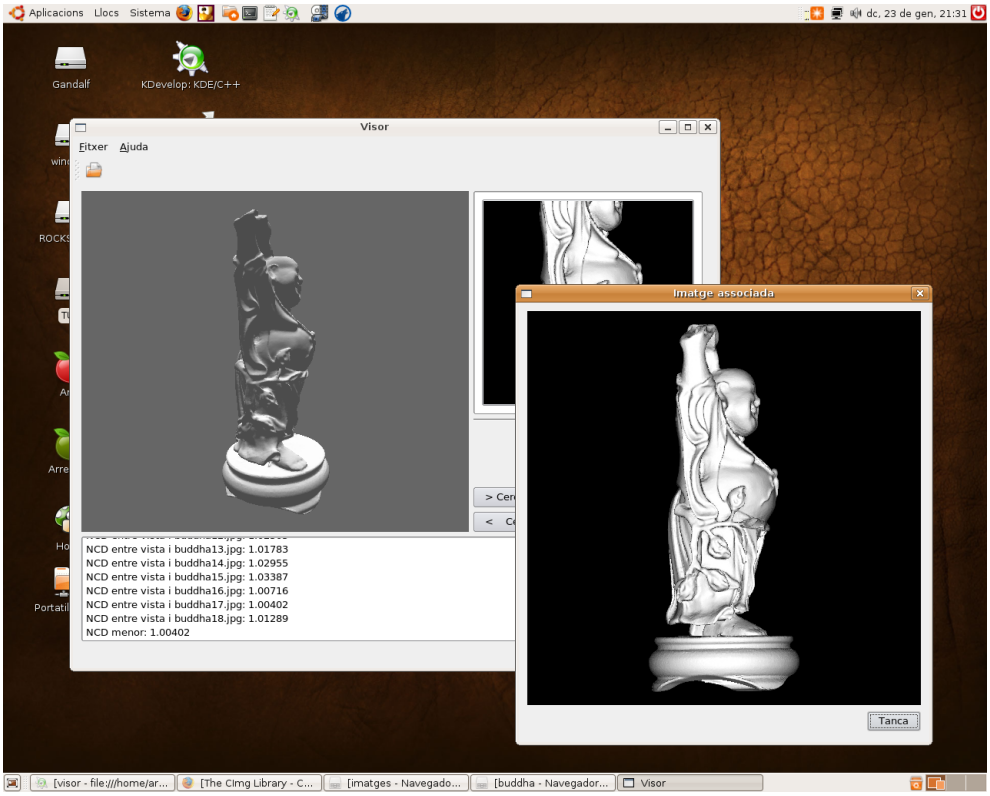


Figura 37: Resultat de la cerca d'imatges feta per *NCD* amb el model en posició lateral

Tercera prova

Imatge lateral del model, cerca de la perspectiva més similar utilitzant 62 càmeres.

Resultat esperat: L'aplicació hauria de triar una de les càmeres que enfoquen al buda lateralment mirant cap a la dreta.

Resultat obtingut amb *MI*:

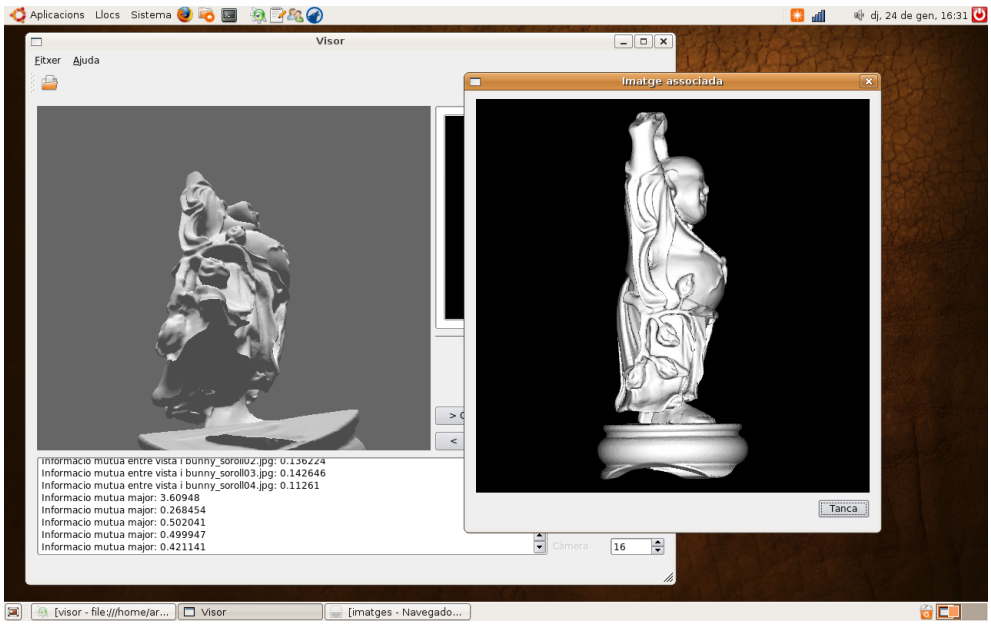


Figura 38: Perspectiva escollida per *MI* donada la imatge lateral del model

Tot i que potser no és la perspectiva més exacte, és una elecció força encertada.

Resultat obtingut amb *NCD*:

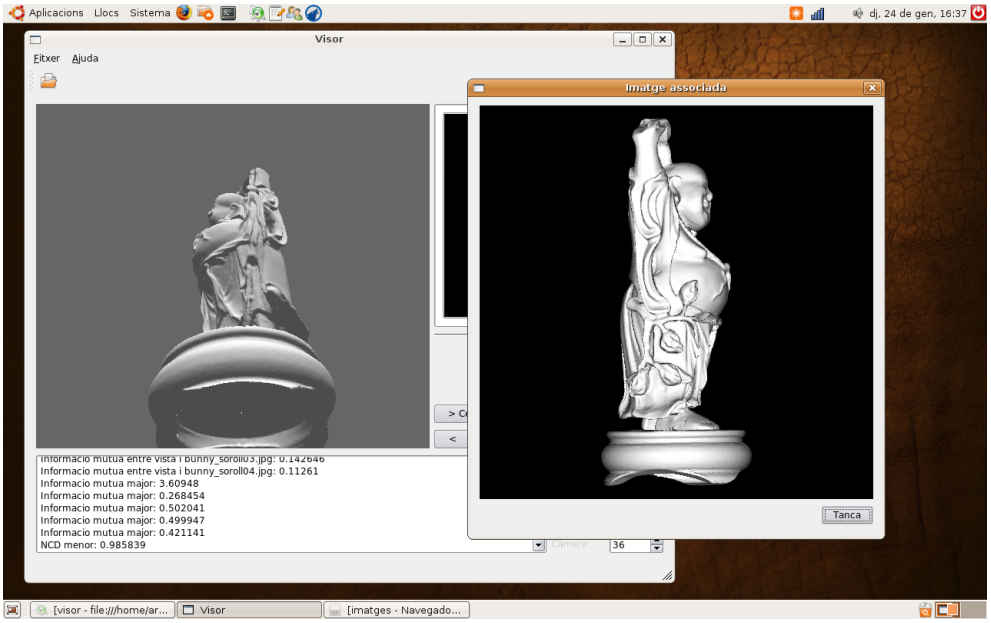


Figura 39: Perspectiva escollida per *NCD* donada la imatge lateral del model

En aquest cas curiosament s'ha escollit la perpectiva oposada. Al basar-se en la mida de les imatges, la *NCD* no discrimina entre una orientació i la inversa.

Quarta prova

Canvi de model, ús del dofí en una posició no del tot frontal.

Resultat esperat: D'entre totes les imatges de budes, els algorismes haurien d'escollir la del dofí.

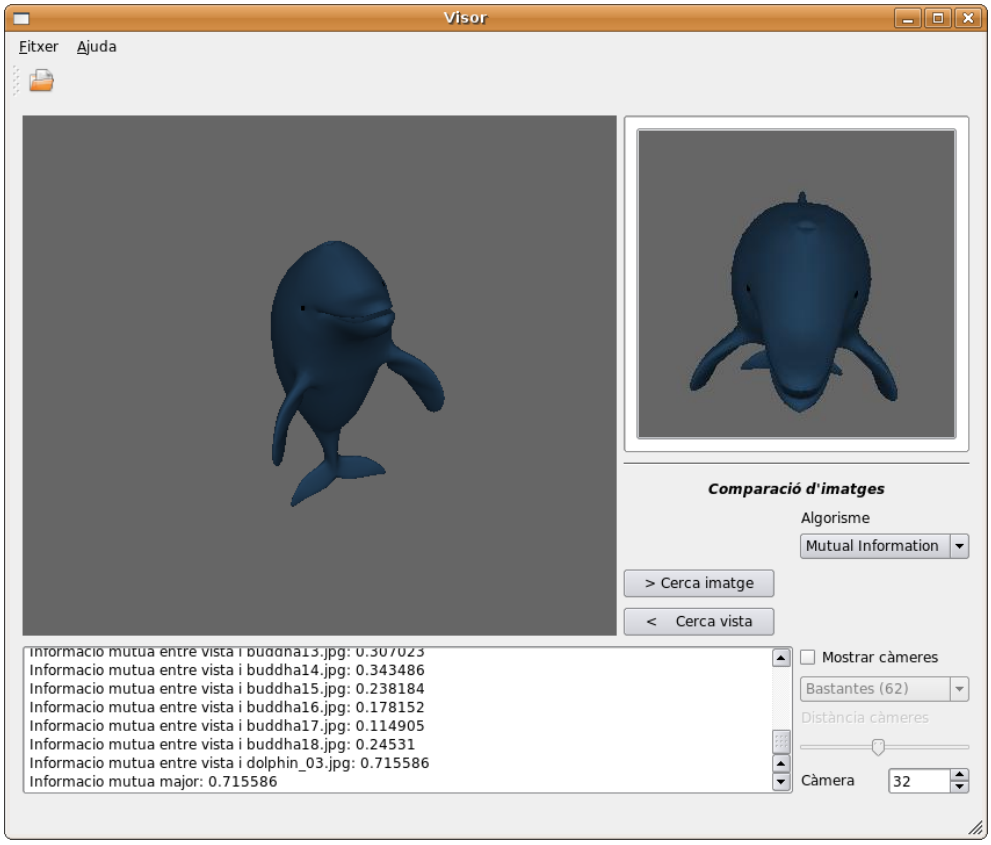


Figura 40: Escollint la imatge del dofí entre les imatges de budes

Els dos algorismes escolleixen correctament la imatge del dofí. En el cas de la *MI* s'obté un valor superior a 0,7 mentre que cap altre imatge arriba al 0,4. Fent servir la *NCD* el valor obtingut és de 0,95 mentre que la resta de valors superen l'1.

La conclusió final seria que els algorismes van ben encaminats, sovint donen resultats satisfactoris, però caldria perfeccionar-los. Estudiant detingudament els comportaments inesperats es podrien detectar flaqueses i

aportar-hi solucions. A més, afegint nous algorismes i buscant convinacions entre ells segurament s'obtindrien resultats encara millors.

4.3 Treball futur

Al tractar-se sobretot d'un projecte d'investigació, es podria considerar treball futur qualsevol evolució de l'aplicació que faci ús de l'experimentació portada a terme. A més, les conclusions extretes donen peu a una sèrie de millores que es podrien dur a terme ja sigui per aportar més eficiència o per obtenir millors resultats, i que podrien resultar necessàries depenent de la funcionalitat que se li volgués donar a l'aplicació.

4.3.1 Millores en la visualització gràfica

OpenGL ofereix més recursos que els que hem utilitzat per millorar el pintat de la geometria, alguns d'ells són:

- *Element Buffer Objects*: Són equivalents als *VBOs* però treballen amb vectors d'índexs. Permeten que els polígons comparteixin vèrtexs.
- Pintat dels *VAs* amb la instrucció *glDrawElements()* i fent servir el mode GL_TRIANGLE_STRIP: Aquesta funció pinta *VAs* que enlloc de contenir els vèrtexs, contenen índexs a cadascún dels vèrtexs. El mode GL_TRIANGLE_STRIP fa que donada una llista de vèrtexs, els tres primers s'interpretin com a un triangle, el segon, tercer i quart com a un altre, i així successivament.

  Aquest pintat és més ràpid i a més no es repeteixen els vèrtexs que formen part de més d'un triangle.
- *Display Lists*: Consisteix en agrupar comandes *OpenGL* que es guarden en memòria de la targeta gràfica per a un ús futur. Posteriorment s'executen totes les instruccions de la llista amb una sola comanda evitant així el nombre de crides a la llibreria.

El seu ús però comporta uns quants inconvenients que s'han de valorar a priori: únicament serveixen per geometria estàtica, les comandes ocupen espai a memòria i el seu ús excessiu pot portar problemes, i les llistes es compilen abans d'emmagatzemar-se, cosa que vol temps.

- *VAs* entrellaçats: Dins d'un mateix *VA* es barreja informació de més d'un atribut dels vèrtexs, com color i coordenades. En alguns casos dóna millors resultats tot i que no sempre.
- Més d'un *VA* per atribut: Si els *VAs* tenen una mida massa gran poden repercutir en l'eficiència en el render. Pot resultar més interessant crear més d'un vector de cada tipus amb mides menors.

Per poder visualitzar els models amb els seus colors reals caldria implementar també la correcta il·luminació a través de shaders o de textures com ja s'ha comentat.

4.3.2 Millores en la comparació d'imatges

Tant l'algorisme de *Mutual Information* com el de la *NCD*, són sensibles a les dimensions de les imatges. Per això ens hem vist obligats a fer que les dues imatges a comparar en cada moment tinguessin la mateixa mida. En els casos en que les imatges s'han de redimensionar, el procés pot distorsionar el resultat final ja que la imatge o bé es deforma o s'ha de retallar o se l'hi han d'afegir píxels.

Caldria estudiar la manera òptima de redimensionar les imatges i veure quins efectes té en els resultats.

A part dels algorismes que s'han vist durant el projecte, també es podrien estudiar i implementar altres algorismes per mirar d'obtenir uns resultats més òptims per a l'aplicació. Sabem per exemple que fer comparacions amb imatges de contorns pot aportar bons resultats. Si a més l'algorisme que treballi a partir d'aquestes imatges no és sensible a escalats i canvis de

perspectiva, segurament observaríem una millora considerable en la qualitat de les comparacions.

També existeix la opció de combinar més d'un algorisme perquè un pugui compensar els errors de l'altre. Això evidentment faria augmentar el cost i s'hauria de valorar si aporta millores suficients.

Referent a l'eficiència, la comparació d'imatges té un cost elevat, i si es té en compte que cada acció de l'usuari requereix moltes comparacions, el procés pot resultar molt feixuc. Caldria per una banda estudiar bé els costos dels algorismes implementats i veure en què es poden millorar, i per altra idear algun mètode en que es pogués avançar feina mentre el programa està inactiu o guardar resultats intermedis que es necessitin més endavant.

Concretament en el cas de la *NCD*, l'ús que es fa de la llibreria *bzip2* és a alt nivell, cosa que comporta lectura i escriptura de fitxers a disc. Fent servir les funcions de més baix nivell que treballen directament amb la memòria, s'evitaria l'entrada sortida i es guanyaria en eficiència.

4.3.3 Evolució de l'aplicació

Un cop realitzades les millores pertinents i havent desenvolupat uns algorismes de comparació d'imatges prou efectius, l'últim pas seria donar-li una finalitat útil a l'aplicació.

Característiques d'una possible aplicació futura basada en el projecte realitzat:

- Visualitzador de models 3D, amb una llibertat de moviments superior i amb possibilitat de situar les càmeres sobre el model d'una manera menys restringida.
- Capacitat per interpretar altres formats de models 3D.
- Obtenció automàtica d'imatges de la Web.

- Adquisició d'informació textual referent als models.
- Cerca automàtica, entre les imatges obtingudes, de les que es corresponen amb el model visualitzat.
- Registre més ampli de les associacions fetes entre models i imatges. Capacitat d'etiquetar els models amb més d'una imatge.

L'aplicació podria servir per il·lustrar mitjançant escenes 3D i imatges, peces de museu, edificis de ciutats, etc. i podria fer la funció de catàleg o d'aplicació informativa.

5 Manual d'utilització de l'aplicació

L'aplicació consisteix en un executable anomenat"visor" i d'una sèrie de carpetes. La carpeta "models" conté els models 3D que es visualitzaran, "imatges" és la carpeta destinada a col·locar les fotografies que es faran servir per a les comparacions, "configs" guarda els arxius de configuració dels models i "tmp" serveix com a directori temporal per alguns dels procediments de l'aplicació.

Amb aquesta senzilla estructuració, l'aplicació no requereix cap instal·lació i es pot moure d'un lloc en un altre sense problemes.

Al obrir el fitxer executable, se'ns presentarà la interfície gràfica del programa i podrem començar a treballar amb ell.

El disseny de la interfície gràfica és ben senzill i es centra en un espai dedicat a la visualització, un quadre on es mostraran les imatges que es comparin, una àrea de missatges i alguns botons. La majoria d'accions es duen a terme mitjançant botons, però algunes tecles també fan funcions secundàries.

Els components de la interfície són els següents:

1. Menú principal amb opcions per obrir models i mostrar l'ajuda.
2. Barra d'eines amb accés a obrir un model.
3. Finestra on es visualitza el model 3D.
4. Àrea de missatges amb informació de les accions que es duen a terme i dels resultats obtinguts.
5. Espai on es visualitzen la imatge associada al model.
6. Selector de l'algorisme a utilitzar per a les comparacions.

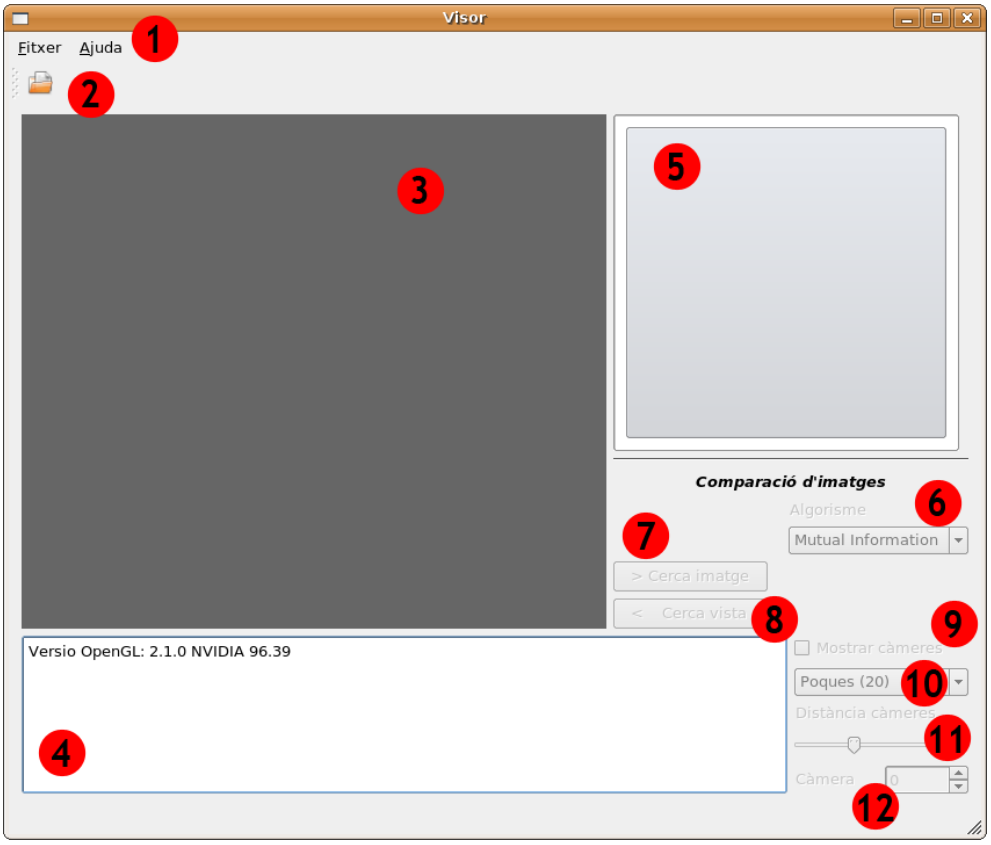


Figura 41: Components de la interfície gràfica

7. Botó per buscar una imatge a partir de la perspectiva del model visualitzat.

8. Botó per escollir una imatge i buscar la perspectiva més semblant a aquesta.

9. Activa o desactiva la visualització de les càmeres al voltant del model.

10. Selector del número de càmeres a fer servir per buscar perspectives.

11. Graduador de la distància de les càmeres al model.

12. Selector de la càmera des de la que es vol visualitzar el model.

El primer que cal fer és carregar un model. Això es pot fer a través del menú "Fitxer | Obre model", per la icona de la barra d'eines o amb la drecera de teclat "Ctrl+O". Escollim el model que vulguem visualitzar i premem *Open*.

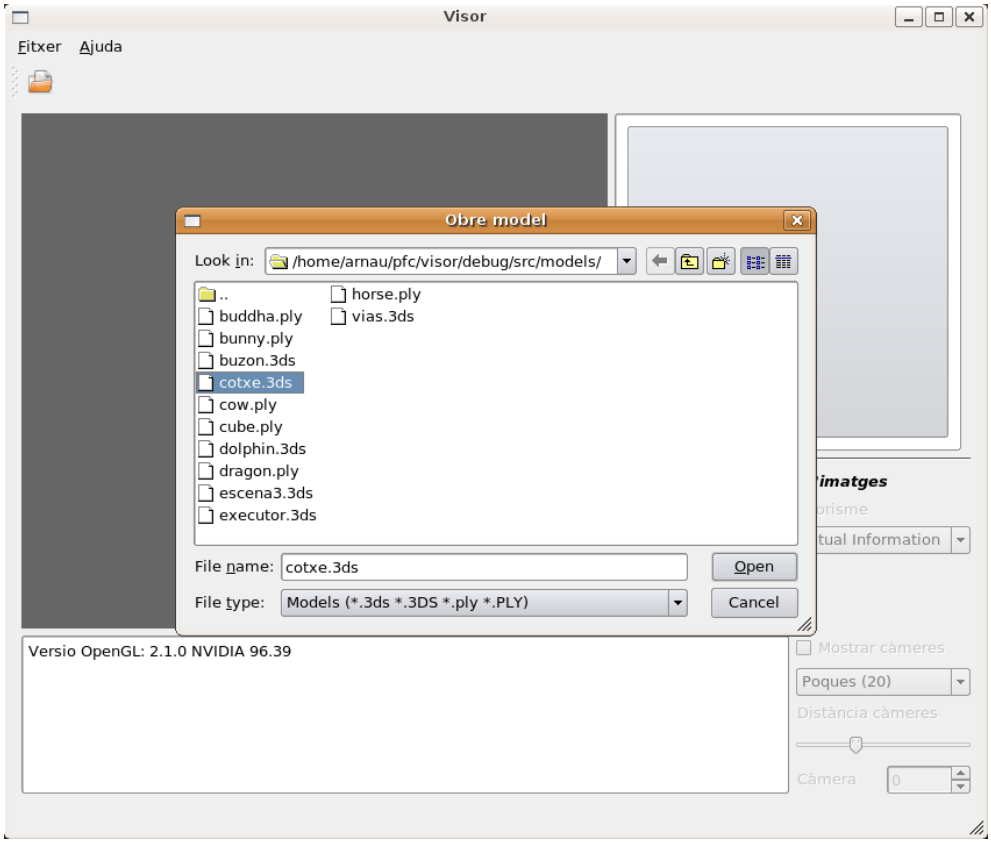


Figura 42: Obrir un model 3D

Un cop obert el model, ens apareixerà dins del visor.

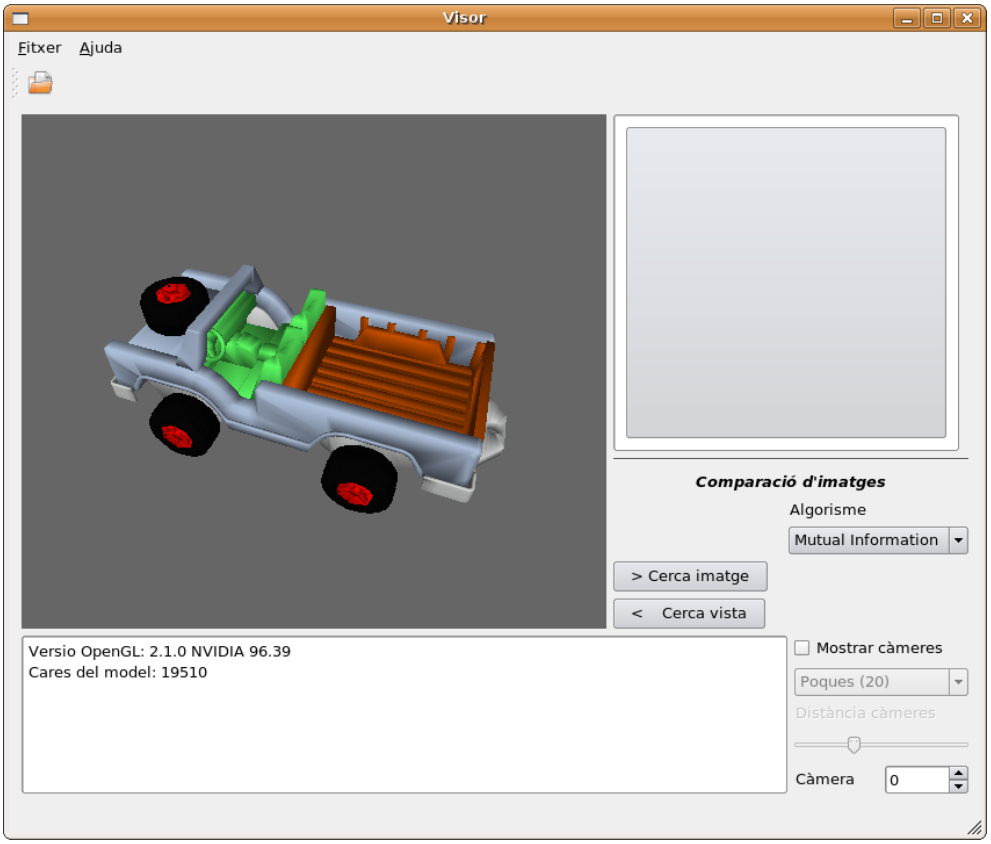


Figura 43: Model carregat

La visualització per defecte es fa utilitzant *VBOs*. Si es vol es pot canviar prement la tecla "o" que ens passarà d'un mètode de pintat a l'altre successivament. També es pot executar el test de freqüència de mostreig amb la tecla "t" per veure el rendiment de cada un dels mètodes.

Amb el model carregat al visor, podem bellugar-lo respecte els seus eixos X i Y amb el botó esquerre del ratolí i respecte l'eix Z amb el botó dret. Els eixos es poden fer visibles o invisibles amb la tecla "e" del teclat. Si premem el botó del mig del ratolí i el movem amunt i avall, ens acostarem o allunyarem del model amb la càmera. I també podem modificar el valor dels plans de retallat anterior i posterior amb les tecles "n/N" i "f/F" res-

pectivament.

Una alta opció per canviar la perspectiva del model és fent servir el selector de càmera.

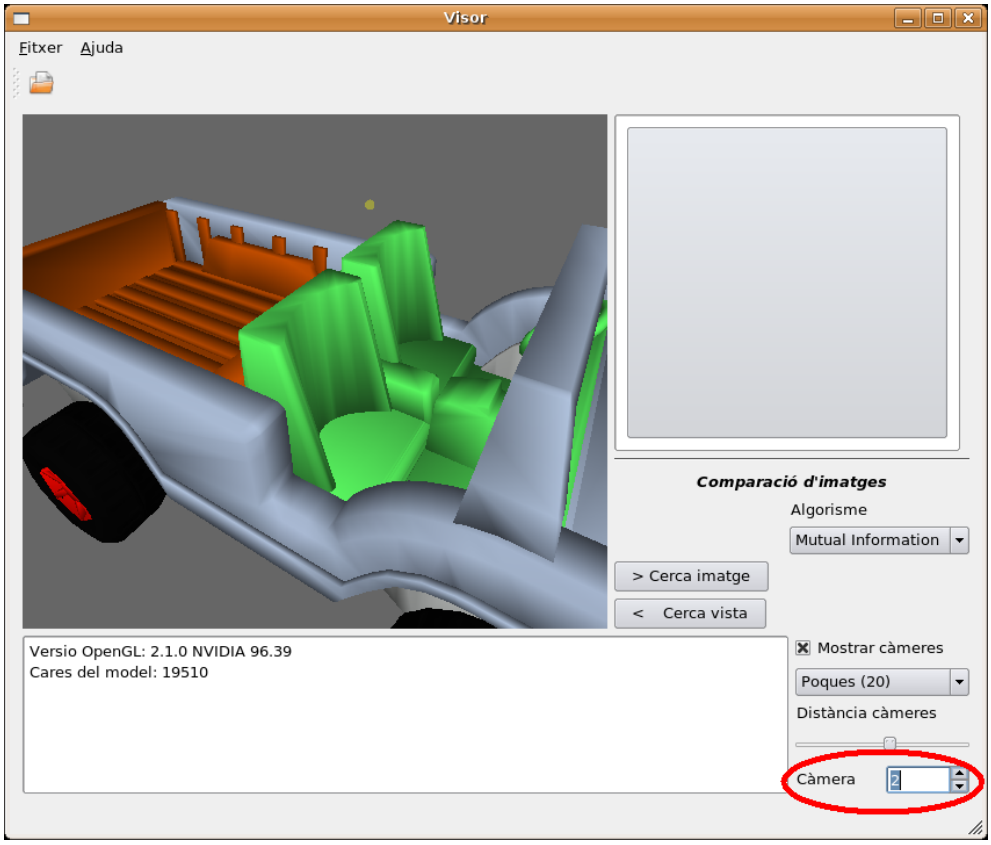


Figura 44: Canvia la càmera des de la que es veu el model

Al canviar entre les diferents càmeres, la visualització s'anirà situant on es troba cadascuna d'elles i podrem tornar a la visualització inicial escollint la càmera 0.

Tenim certa llibertat per modificar els paràmetres d'aquestes càmeres començant per l'activació o desactivació de la seva visualització. Aquesta llibertat ens serà útil a l'hora de voler buscar una perspectiva adient per una imatge escollida.

Podem variar el número de càmeres que es faran servir, poques (20), bastantes (62) o moltes (242).

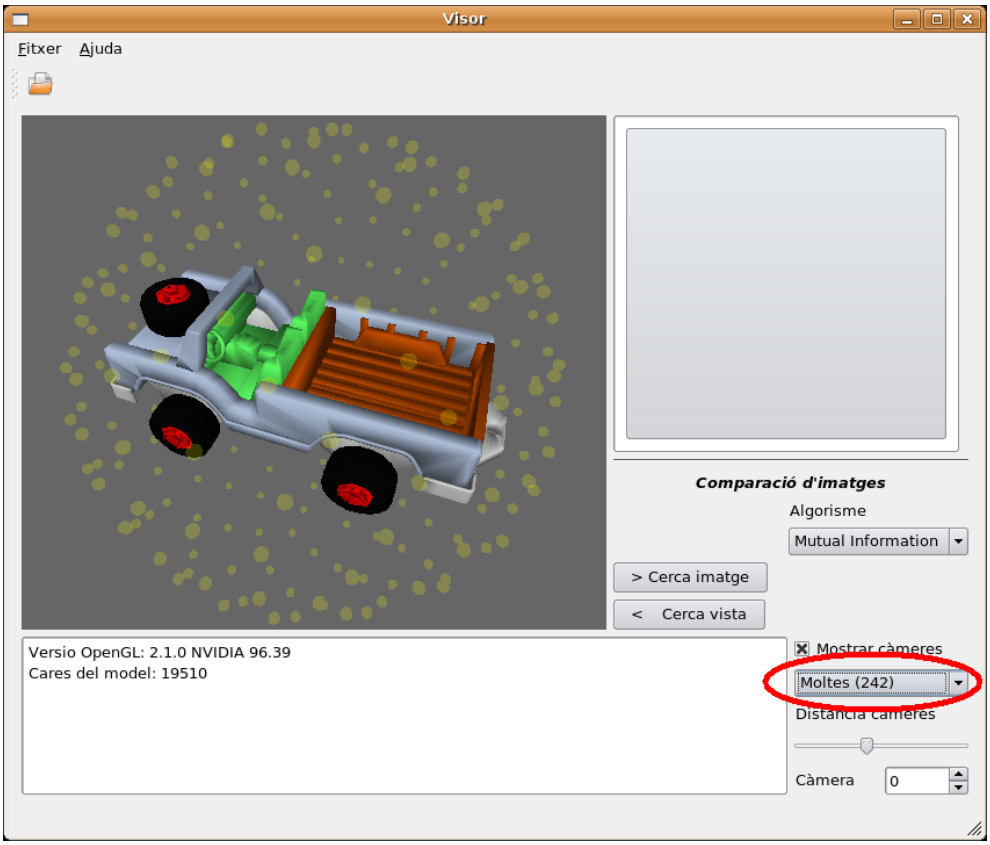


Figura 45: Quantitat de càmeres per captar perspectives del model

També podem variar la distància entre les càmeres i el model amb l'*slider* corresponent.

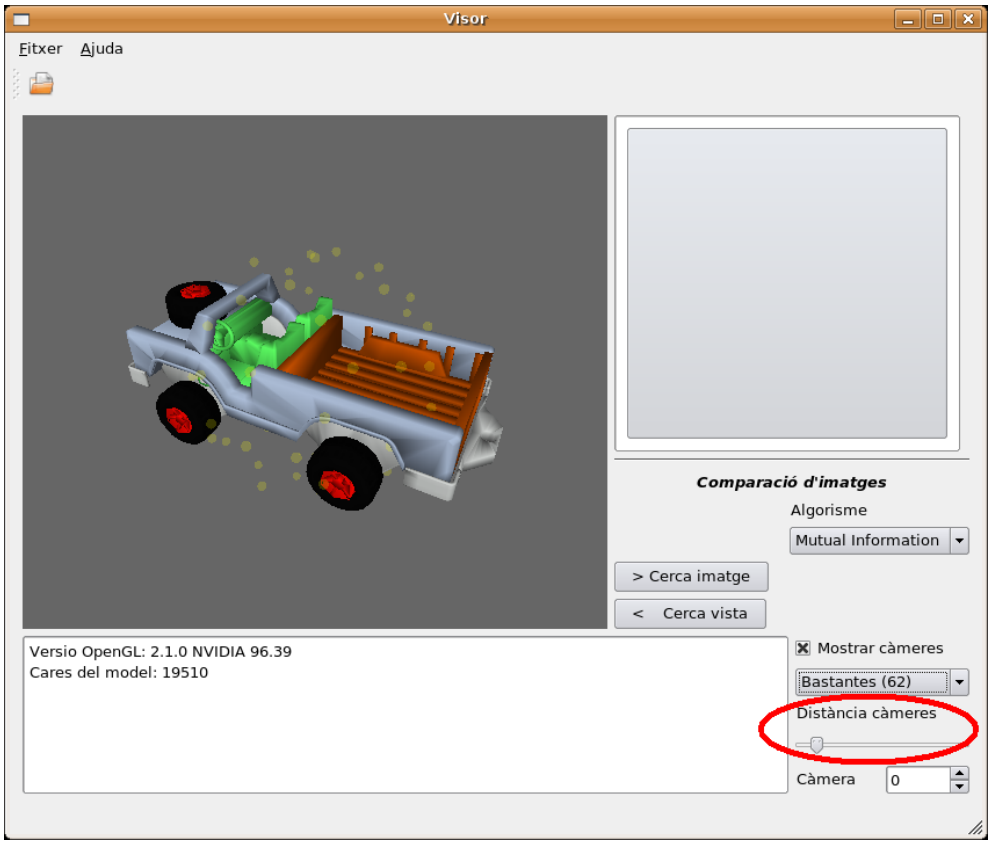


Figura 46: Proximitat de les càmeres amb el model

Un cop escollida una vista del model mitjançant els recursos anterior, ja podem fer una comparació d'imatges per veure quina fotografia ens troba el programa que s'ajusti al nostre model en la perspectiva actual.

Per fer això escollim l'algorisme de comparació que volem fer servir i premerem el botó "> Cerca imatge".

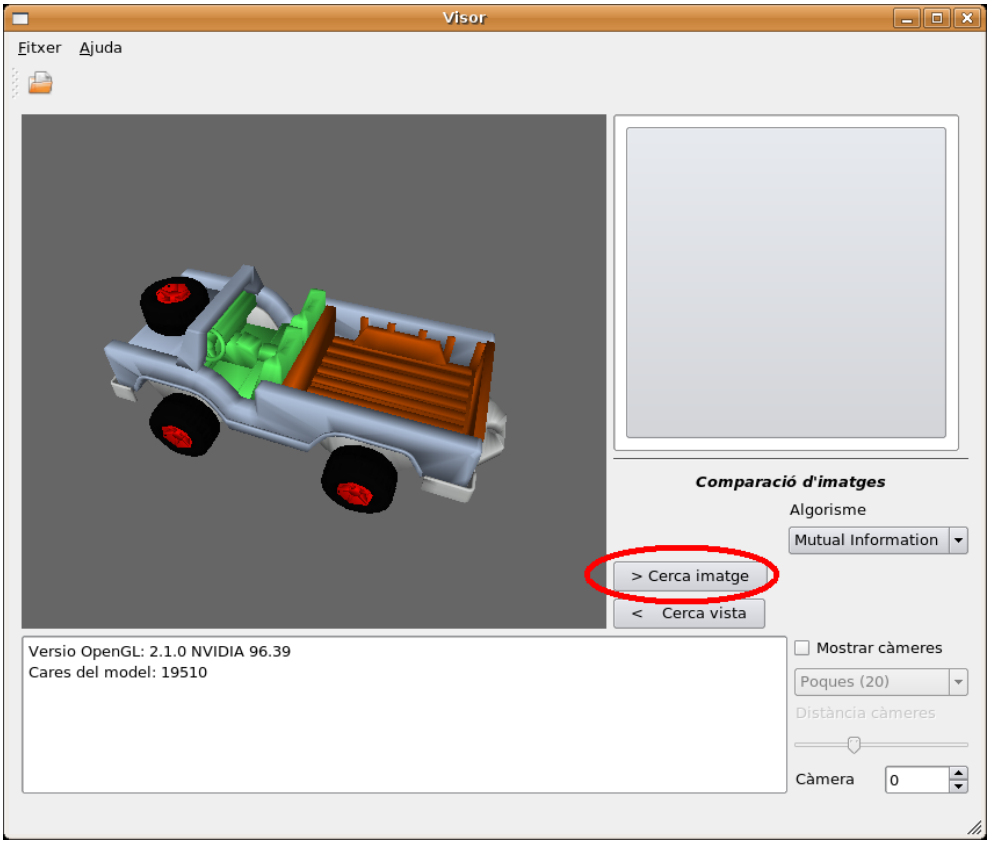


Figura 47: Ordre de buscar la imatge més semblant a la perspectiva del model

Al cap d'una estona el programa ens mostrarà quina és la imatge més semblant que ha trobat, a l'àrea de missatges ens mostrarà els valors numèrics de les comparacions i al requadre destinat a les imatges ens hi carregarà la imatge escollida.

Com que la imatge només es mostra parcialment en el requadre, podem clicar-la per obrir una finestra on se'ns mostri ampliada.

Figura 48: Ampliació de la imatge associada al model

El procés invers és escollir una imatge perque ens trobi la perspectiva del model que més s'hi ajusti. Per fer això escollim quantes càmeres volem que es facin servir per buscar perspectives, si volem ajustem la seva posició i premem el botó "< Cerca vista". Escollim una imatge i esperem que el programa busqui entre les vistes que ofereixen les diferents càmeres quina és la més adient per la imatge escollida.

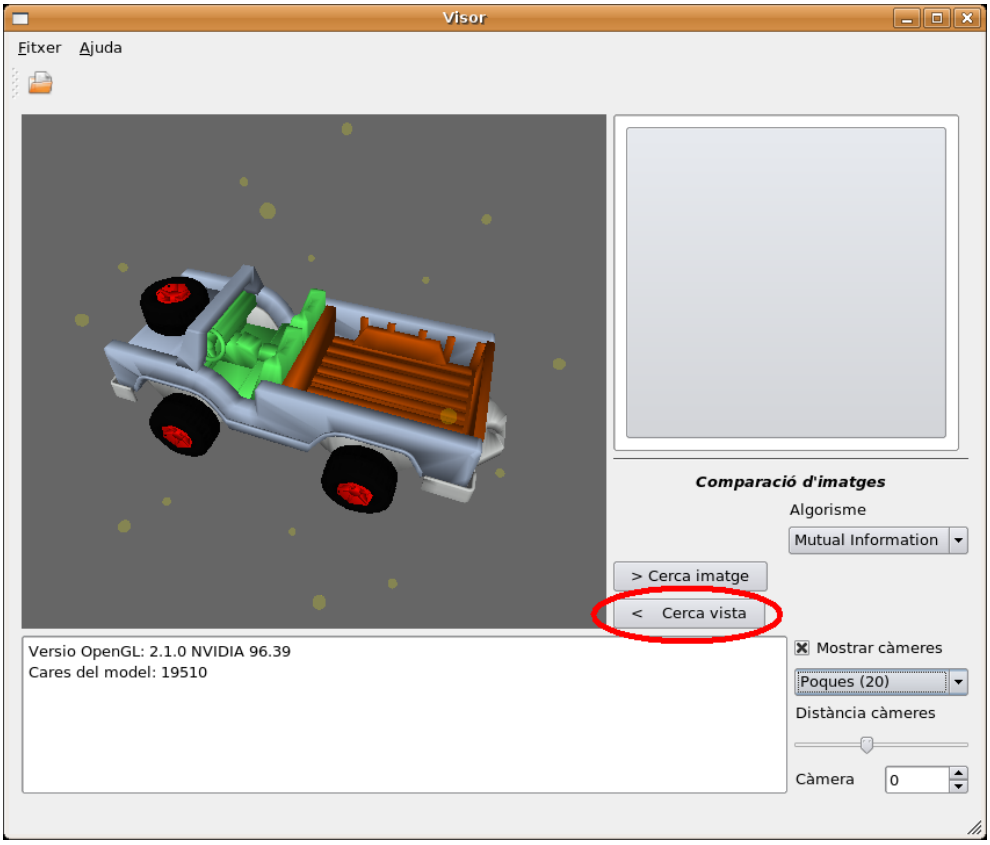


Figura 49: Ordre de buscar la perspectiva més semblant a una imatge

Si tenim algún dubte de les opcions per manipular el model podem recórre a l'ajuada amb les tecles "F1", "h" o a través del menú, on trobarem també la finestra "Quant a..." amb informació de l'aplicació i l'autor.

Figura 50: Finestra d'ajuda

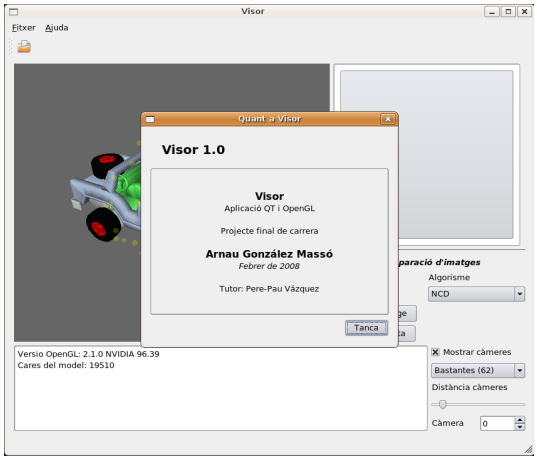


Figura 51: Informació de l'aplicació i l'autor

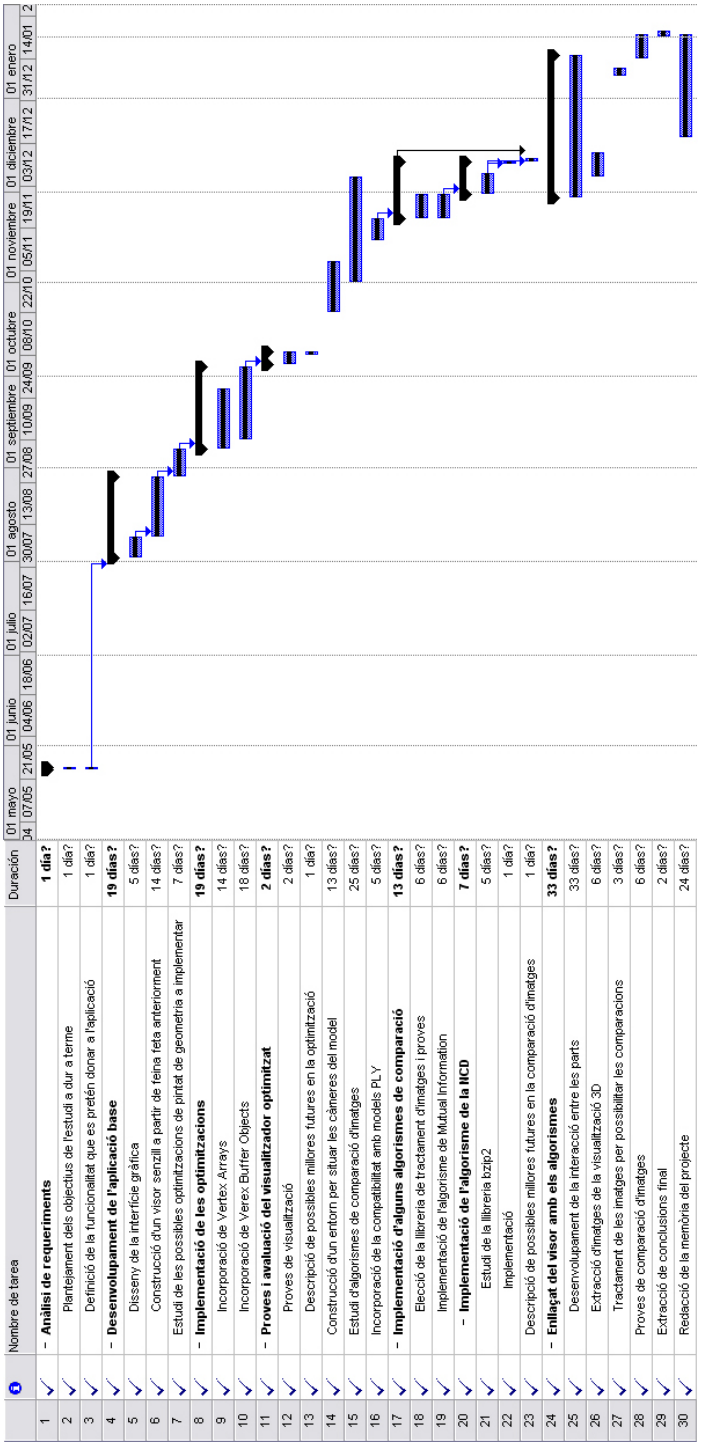
Per sortir de l'aplicació n'hi ha prou amb escollir la opció del menú o prémer la creu de la cantonada superior dreta.

6 Planificació

L'elaboració d'aquest projecte ha passat per diverses fases d'estudi, desenvolupament i proves que ens han portat al resultat final. Cada fase d'aquestes ha quedat dividida en tasques específiques que es descriuen a continuació:

- Anàlisi de requeriments
  - Plantejament dels objectius de l'estudi a dur a terme
  - Definició de la funcionalitat que es pretén donar a l'aplicació
- Desenvolupament de l'aplicació base
  - Disseny de la interfície gràfica
  - Construcció d'un visor senzill a partir de feina feta anteriorment
- Estudi de les possibles optimitzacions de pintat de geometria a implementar
- Implementació de les optimitzacions
  - Incorporació de Vertex Arrays
  - Incorporació de Verex Buffer Objects
- Proves i avaluació del visualitzador optimitzat
  - Descripció de possibles millores futures en la optimització
- Construcció d'un entorn per situar les càmeres del model
- Estudi d'algorismes de comparació d'imatges
- Incorporació de la compatibilitat amb models PLY
- Implementació d'alguns algorismes de comparació
  - Elecció de la llibreria de tractament d'imatges i proves

  - Implementació de l'algorisme de Mutual Information
  - Implementació de l'algorisme de la NCD
    * Estudi de la llibreria bzip2
- Descripció de possibles millores futures en la comparació d'imatges
- Enllaçat del visor amb els algorismes
  - Extracció d'imatges de la visualització 3D
  - Tractament de les imatges per possibilitar les comparacions
- Proves de comparació d'imatges
- Extracció de conclusions final
- Redacció de la memòria del projecte



		Nombre de tarea	Duración
1	✓	**- Anàlisi de requeriments**	**1 día?**
2	✓	Plantejament dels objectius de l'estudi a dur a terme	1 día?
3	✓	Definició de la funcionalitat que es pretén donar a l'aplicació	1 día?
4	✓	**- Desenvolupament de l'aplicació base**	**19 días?**
5	✓	Disseny de la interfície gràfica	5 días?
6	✓	Construcció d'un visor senzill a partir de feina feta anteriorment	14 días?
7	✓	Estudi de les possibles optimitzacions de pintat de geometria a implementar	7 días?
8	✓	**- Implementació de les optimitzacions**	**19 días?**
9	✓	Incorporació de Vertex Arrays	14 días?
10	✓	Incorporació de Verex Buffer Objects	18 días?
11	✓	**- Proves i avaluació del visualitzador optimitzat**	**2 días?**
12	✓	Proves de visualització	2 días?
13	✓	Descripció de possibles millores futures en la optimització	1 día?
14	✓	Construcció d'un entorn per situar les càmeres del model	13 días?
15	✓	Estudi d'algorismes de comparació d'imatges	25 días?
16	✓	Incorporació de la compatibilitat amb models PLY	5 días?
17	✓	**- Implementació d'alguns algorismes de comparació**	**13 días?**
18	✓	Elecció de la llibreria de tractament d'imatges i proves	6 días?
19	✓	Implementació de l'algorisme de Mutual Information	6 días?
20	✓	**- Implementació de l'algorisme de la IICD**	**7 días?**
21	✓	Estudi de la llibreria bzip2	5 días?
22	✓	Implementació	1 día?
23	✓	Descripció de possibles millores futures en la comparació d'imatges	1 día?
24	✓	**- Enllaç del visor amb els algorismes**	**33 días?**
25	✓	Desenvolupament de la interacció entre les parts	33 días?
26	✓	Extracció d'imatges de la visualització 3D	6 días?
27	✓	Tractament de les imatges per possibilitar les comparacions	3 días?
28	✓	Proves de comparació d'imatges	6 días?
29	✓	Extracció de conclusions final	2 días?
30	✓	Redacció de la memòria del projecte	24 días?

Figura 52: Planificació

7 Anàlisi econòmic

Durant l'evolució del projecte s'han jugat els rols d'analista i de programador per fer l'estudi d'optimitzacions i algorismes de comparació d'imatges, i pel desenvolupament de l'aplicació respectivament. La dedicació ha estat d'una persona, 4 hores al dia.

Amb el càlcul d'hores extret de la planificació i uns sous estàndards per als rols, els costos de recursos són els següents:

Rol	**Hores invertides**	**Euros / Hora**
Analista	180	20
Programador	440	8

Taula 9: Costos dels recursos

El desenvolupament no ha requerit llicències de programari ja que s'ha treballat amb codi lliure: *Linux*, *KDevelop*, *OpenGL*, *CImg Library*, *Bzip2*.

Si que cal afegir als costos el material informàtic utilitzat. Un ordinador i una targeta gràfica, ambdós de gamma mitja-alta.
Ordinador: 1.000 €
Targeta gràfica: 150 €

El cost total és el següent:
Recursos humans: 180h x 20 €/h + 440h x 8 €/h = 7.120 €
Material informàtic: 1.000 € + 150 € = 1.150 €
Total: 7.120 € + 1.150 € = **8.270** €

8 Glossari

- **Escena 3D**: Espai 3D que pot estar format per geometria, comportament de llums, textures, materials i animació. Pretén representar ambients i estructures físiques versemblants.

- **Framebuffer**: Dispositiu virtual del sistema operatiu que es presenta a les aplicacions com un arxiu o un bloc de memòria reservat i que pot ser accedit per lectura o escriptura pels processos. Qualsevol escriptura en aquest arxiu o zona de memòria modifica directament les imatges visualitzades en el dispositiu de vídeo, perquè d'aquesta manera els programes puguin mostrar informació en pantalla sense preocupar-se dels detalls d'implementació ni de la interacció real entre l'ordinador i el dispositiu de vídeo.

- **Imatges multimodals**: Imatges que capten informació diferent de la mateixa cosa. Un exemple seria una fotografia presa amb una càmera de visió nocturna i la mateixa fent servir una càmera sensible a la calor.

- **Llibreria gràfica**: Conjunt de funcions i procediments que serveixen per construir aplicacions gràfiques en 2D i/o 3D.

- **Llum ambient**: Llum general present en un lloc. La seva font es troba tant lluny que es pot considerar que no prové d'un lloc precís.

- **Llum difusa**: Llum escampada de manera uniforme al reflectir en un cos. Molt característica dels cossos mates.

- **Llum especular**: És el tipus de llum que es refereix a les reflexions en cossos brillants. Ve molt condicionada per la posició de la font de llum.

- **Malla de triangles**: Superfície tridimensional construïda com una col·lecció de cares de 3 vèrtexs. Els vèrtexs es defineixen mitjançant les seves coordenades posicionades en un espai virtual.

- **Màquina d'estats**: Sistema la configuració del qual depèn no només de les senyals d'entrada sinó també de les senyals de sortida i estat anteriors.

- **Material**: En termes de gràfics 3D, són les propietats associades a la geometria dels models 3D que es refereix a les seves propietats òptiques, és a dir, com respon la geometria a nivell de color davant de la llum.

- **Model 3D**: Conjunt de polígons que representa un objecte, ésser, edifici, etc. Existeixen múltiples maneres d'estructurar els models i poden portar informació addicional a la geometria com propietats de color o reacció a la llum.

- **Modelview**: Matriu utilitzada per *OpenGL* per guardar les transformacions de vista i del model a visualitzar. S'encarrega de transformar les coordenades originals de l'objecte en coordenades d'ull, les que es mostraran per pantalla.

- **Normal**: En termes matemàtics, és el vector perpendicular a una superfície. La normal d'una cara es calcula com el producte vectorial de dues de els seves arestes. La normal d'un vèrtex és la mitja de les normals de les cares que comparteixen aquest vèrtex.

- **OpenGL**: Especificació estàndard guiada pel consorci independent *OpenGL Architecture Review Board*. Existeixen diverses implementacions d'aquesta llibreria per a llenguatges de programació diferents.

- **Plans de retallat**: Plans virtuals que delimiten la regió a visualitzar d'una escena 3D. Si s'utilitza una càmera perspectiva, només es fan servir el pla anterior, situat entre la càmera i el model, i el posterior, situat darrera del model. L'angle de visió i la relació d'aspecte acoten la resta de l'espai. Per a càmeres axonomètriques també s'han de definir els plants de retallat laterals, superior i inferior. El prisma que construeixen aquests plans és l'espai visualitzat.

- **Primitiva**: Referent a la construcció de geometria, es refereix a una forma geomètrica simple com un cub, una esfera, un cilindre, o més simple encara un punt o una recta.
- **Refresc**: Repintat del que s'està visualitzant. Cada vegada que hi ha una modificació en una escena 3D, cal fer un refresc per que es mostrin els canvis.
- **Relació d'aspecte**: Proporció entre amplada i alçada d'una imatge. Es calcula dividint l'amplada per l'alçada de la imatge visible en pantalla.
- **Renderització**: Generació d'una imatge plana des d'un model en tres dimensions. Sol tractar-se d'un procés complex ja que cal tenir en compte molts factors com la il·luminació que requereix una gran quantitat de càlculs.
- **Shader**: Conjunt d'instruccions utilitzat pels recursos gràfics principalment per dur a terme efectes de renderitzat. Els *shaders* s'utilitzen per permetre als dissenyadors d'aplicacions 3D a programar el processador del maquinari gràfic i guanyar en eficiència.
- **Signals i slots**: Sistema de comunicació entre parts d'un programa. Els *signals*, elements destinats a llençar un avís, es vinculen a accions com prémer un botó i es connecten a *slots*, un tipus concret de funció, definits en una altra part del programa. Els slots porten a terme la seva funció concreta quan el signal al que estan connectats és activat.
- **Textura**: Imatge plana utilitzada per dotar de detall a un model 3D sense necessitat de fer-ho amb geometria. S'aplica sobre la superfície del model i permet augmentar el nivell de detall amb un cost de càlcul molt inferior.
- **Widget**: Terme construït a partir dels termes anglesos finestra (*window*) i giny (*gadget*), i es refereix als elements propis de les interfícies gràfiques com les finestres, els botons, les barres de desplaçament, etc.

Referències

[1] Paul Martz. *OpenGL Distilled*. Addison Wesley Professional, 2006

[2] Mark Segal, Kurt Akeley. *The OpenGL Graphics System: A Specification - Version 2.1*. Silicon Graphics, Inc., 2006

[3] Christophe [Groove] Riccio. *Vertex Buffer Objects*. G-Truc Creations, 2006

[4] Louis Bavoil. *Rendering huge triangle meshes with OpenGL*. University of Utah, 2005

[5] Rudi Cilibrasi, Paul Vitanyi. *Clustering by Compression*. CWI and University of Amsterdam, 2004

[6] Barbara Zitová, Jan Flusser. *Image registration methods: a survey*. El Sevier, 2003

[7] Peter Grünwald and Paul Vitányi. *Shannon Information and Kolmogorov Complexity*. 2004

[8] OpenGL Reference Manual
http://www.opengl.org/documentation/blue_book/

[9] OpenGL Programming Guide
http://www.opengl.org/documentation/red_book/

[10] OpenGL Man Pages
http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/

[11] Forums OpenGL
http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=cfrm

[12] Lib3ds Data Structure
http://lib3ds.sourceforge.net/classes.html

[13] Models PLY
http://www.cc.gatech.edu/projects/large_models/index.html

[14] Llibreria CImg
http://cimg.sourceforge.net

[15] SIFT
http://vision.ucla.edu/ vedaldi/code/siftpp/siftpp.html
http://www.cs.ubc.ca/ lowe/keypoints/

[16] NCD
www.complearn.org/ncd.html

Índex de figures